Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference

Faranak Heidarian Dehkordi

ISBN 9789461913333
IPA dissertation series 2012-06

Typeset with LaTeX 2ε
Published and printed by: Ipskamp Drukkers, Nijmegen
Cover design: Seyyed Hamed Hashemi

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

NWO
Nederlandse Organisatie voor Wetenschappelijk Onderzoek

This research was supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.064.610.

Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op donderdag 5 juli 2012
om 10:30 uur precies
door

Faranak Heidarian Dehkordi

geboren op 24 augustus 1981
te Shahrekord, Iran

Promotor:

Prof. dr. F.W. Vaandrager

Manuscriptcommissie:

Prof. dr. J.J.M. Hooman

Prof. dr. B. Jonsson, Uppsala Universitet, Uppsala

Prof. dr.J.C. van de Pol, Universiteit Twente

Dr. A. Silva

Dr. M. Verhoef, Chess eT International B.V., Haarlem

Preface

It would not have been possible to write this doctoral thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

Above all, I would like to thank my advisor Frits Vaandrager for his kind support, advice, and patience. I gratefully acknowledge the financial support of the Netherlands Organization for Scientific Research (NWO). I appreciate the academic and technical support of the Radboud University Nijmegen and its staff. I am most grateful to people of Chess eT International B.V., specially, Marcel Verhoef, Frits van der Wateren, and Bert Bos for arranging fruitful meetings at Chess, providing the details of MyriaNed and Median synchronization protocols, and being always available for answering the questions. I am also thankful to Pieter Anemaet, Fasika Assegei, and Venkat Iyer. I thank my manuscript committee, Jozef Hooman, Bengt Jonsson, Jaco van de Pol, Alexandra Silva, and Marcel Verhoef for reading my thesis, and providing me with useful comments. I acknowledge Julien Schmaltz for the very fruitful collaboration on analysis of MyriaNed protocol, and useful discussions that I had with him. Thanks also to the anonymous reviewers of my contributed papers for suggestions that helped improving the papers.

I thank my colleagues in the computer science department including Fides Aarts, Ingrid Berenbroek, Jasper Berendsen, Irma Haerkens, Arjen Hommersom, Georgeta Igna, David Jansen, Harco Kuppens, Martijn Lappenschaar, Bas Lijnse, Steffen Michels, Agnes Nakakawa, Olha Shkaravska, Jan Tretmans, Fiona Tulinayo, John van Groningen, Maria van Kuppeveld, Thomas van Noort, Wenyun Quan, Marina Velikova, Freek Verbeek, and Ilona Wilmont for everything that I learned from them, for friendship and the time that we spent together.

I specially thank my parents who as always, have given me their unequivocal support, for which my mere expression of thanks does not suffice. Last but not least, I am most personally indebted to my husband, and best friend Ali. We started our PhD studies quite at the same time. Now that we made our journey together, I cannot wait to see what is in store for us next.

Nijmegen, April 2012

Contents

Chapter 1

Introduction

In daily life, computers are omnipresent. As a matter of fact, we are surrounded by computers, which we use to search for or share information on the Internet, to communicate via email and social networks, to keep track of financial data, or to express ourselves by writing blogs or by podcasting our voices. The vast majority of computers in use, however, are much less visible: They run the engine, brakes, seat belts, airbags, and audio system in our cars and control aircrafts and trains. They digitally encode our voices and construct radio signals to send from our cell phones to a base station. They control our microwave oven, refrigerator, and dishwasher. They command robots on a factory floor, power generation in a power plant, processes in a chemical plant, and traffic lights in a city. They search for microbes in biological samples, construct images of the inside of a human body, and measure vital signs. These less visible computers are called *embedded systems*.

Embedded systems are redefining how we perceive and interact with the physical world. Since the beginning of the 21st century, our society has witnessed a great increase in technological advances, and this has affected our lifestyle in various ways. On the one hand the young toddlers learn to deal with a wide variety of tools and devices, from digital toys to electronic games, as a part of the world in which they are supposed to live; on the other hand, the elderly adapt themselves appropriately to interact with every-day-coming new machines which are to make life easier. We go jogging with an iPod in our pocket which not only plays music for us but also computes our speed.

Strong demand for adding more features to software applications has led to much larger and more complex embedded systems. While mission-critical embedded applications raise obvious reliability concerns, unexpected or premature failures in even noncritical applications such as game boxes and portable video players can erode a manufacturer's reputation and greatly diminish acceptance of new devices. The advent of more sophisticated embedded systems that support more powerful functions have brought reliability concerns to the forefront. The design of reliable systems requires assuring that the system never moves through

a dangerous state and verification and validation is the key. Indeed, *verification* and *validation* (V & V) is the process of checking that a software system meets its specifications and that it fulfills its intended purpose. Testing is used in association with verification and validation to disclose possible errors of the system, whereas model checking and theorem proving are using mathematical methods to prove the correctness of the model of the system.

In almost every engineering discipline, models have been used to manage system complexity. Developers have employed them as reusable and analyzable artifacts to bridge the conceptual gap between requirements and target-system implementations. *Model-based development* (MBD) relies on the use of explicit models to describe development activities and products. Applying MBD in embedded-system applications development encourages practitioners to use testing techniques that take another track than that of traditional techniques. The construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Automated support for constructing behavioral models of implemented components would therefore be extremely useful.

This dissertation approaches V & V of embedded systems with two different analogies: in the first part, modeling and verification of a real-world case-study provided by the Chess eT International B.V. is described, whereas the second part investigates automata learning (automatic modeling of systems) using abstraction refinement.

This introduction is organized, as follows. In section 1.1, embedded systems are briefly introduced and a short account of their V & V challenges is presented, then in section 1.2, Chess case is informally described. Afterwards, section 1.3 presents automata learning and describes the problem which is explained in the second part of the thesis. Finally, the structure of this thesis is outlined in section 1.4.

1.1 V & V of Embedded Systems

Embedded systems are information processing systems embedded into enclosing products [60]. In other words, embedded systems are integrated hardware/software systems built into devices that are not necessarily recognized as computerized devices or computers. However, these embedded processing units control and actually define the functionality and quality of these devices. Embedded systems are typically not monolithic, but consist of multiple processing units, connected through wired or wireless networks. The size of the system components ranges from tiny battery-powered intelligent sensors and actuators, to large multiple-rack computing devices. These distributed or networked systems come with a large number of common characteristics, including real-time constraints, and dependability as well as efficiency requirements and strict resource constraints, and ranging from limited energy supply, memory and processing power to space and weight constraints. The

design of embedded systems is therefore intrinsically a multi-disciplinary activity, requiring skills from computer science, electronics and mechatronics and control, along with a thorough understanding and interaction with the application field. See [44].

For embedded systems, the link to physics and physical systems is rather important. Accordingly, the term *cyber-physical systems* (CPS) was coined by Lee [61] to refer to the integration of computation with physical processes. Cyber-physical systems are embedded computers and networks monitoring and controlling the physical processes, usually with feedback loops where physical processes affect computations and vice versa. Unlike more traditional embedded systems, a full-fledged CPS is typically designed as a network of interacting elements with physical input and output instead of as standalone devices [62]. This notion is closely tied to concepts of robotics and sensor networks.

In connection with the concept of cyber-physical systems, embedded systems are defined as integrations of software and hardware where the software reacts to sensory data and/or issues commands to actuators. The physical system is an integral part of the design and the software must be conceptualized to act jointly with that physical system. Physical systems are intrinsically concurrent and temporal. Actions and reactions happen simultaneously and over time, and the metric properties of time are an essential part of the behavior of the system [59]. In embedded software, therefore, time matters and concurrency and interaction with hardware are intrinsic, since embedded software engages the physical world in non-trivial ways (more than keyboards and screens).

Our reliance on embedded systems makes their reliable operation of large social importance. Besides, offering a good performance in terms of response times, processing capacity and the absence of annoying errors is one of the major quality indications. Consequently, there is a crucial need for formalisms, techniques, and tools that enable the efficient design of correct and well-functioning systems despite their complexity, and make it possible to guarantee their correctness, during maintenance. The availability of such tools could contribute to significant savings in the time and the cost of developing and certifying embedded systems. For safety-critical systems, these tools could save lives (e.g., in avionics and military systems).

Prevailing industrial practice in embedded software relies on bench testing for concurrency and timing properties. This has worked reasonably well, because programs are small, and because the software gets encased in a box with no outside connectivity that can alter the behavior of the software. However, applications today demand that embedded systems be feature-rich and networked, so bench testing become inadequate. In a networked environment, it becomes impossible to test the software under all possible conditions, because the environment is not known. Moreover, general-purpose networking techniques themselves make program behavior much more unpredictable.

Formal methods are a particular kind of mathematically-based techniques for

the specification, development and verification of software and hardware systems, where models are created that can be rigorously analyzed using techniques from mathematics and logic that provide objective evidence of well-formedness, soundness and completeness of the model. Formal methods can help detect flaws, and in the process can improve the understanding that a designer has of the behavior of a complex system.

In the coming subsections, a brief description of modeling and verification of embedded systems is presented.

1.1.1 Modeling

Specification is a description of how a system is supposed to behave. Specifications for embedded systems provide models of the system under design (SUD). A *model* is a simplification of another entity, which can be a physical thing or another model [50]. The model contains those characteristics and properties of the modeled entity that are relevant for a given task.

Working with models has a major advantage. Models can have formal properties. We can say definitive things about models. For example, we can assert that a model is deterministic, meaning that given the same inputs it will always produce the same outputs. If our model is a good abstraction of the physical system, then the definitive assertion about the model gives us confidence in the physical realization of the system. Such confidence is hugely valuable, particularly for embedded systems where malfunctions can threaten human lives. Studying models of systems gives us insight into how those systems will behave in the physical world.

Real-life systems are distributed, concurrent systems composed of components. It is therefore necessary to be able to specify concurrency conveniently. Components must be able to communicate and to synchronize. Furthermore, many embedded systems are real-time systems. Therefore, explicit timing requirements are one of the characteristics of embedded systems. The behavior of time-critical systems is typically subject to rather stringent timing constraints. For a train crossing it is essential that on detecting the approach of a train, the gate is closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. For a radiation machine the time period during which a cancer patient is subjected to a high dose of radiation is extremely important; a small extension of this period is dangerous and can cause the patients death. See [65].

Classical finite state machines do not provide information about time. In order to model time, classical automata have been extended to also include timing information. *Timed automata* [4] are essentially automata extended with real-valued variables. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints, i.e. guards on edges, are used to restrict the behavior of the automaton. Uppaal [17] is an integrated tool environment for

modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays etc.).

1.1.2 Verification

Verification is the process of proving or demonstrating that the program correctly complies to its specification. In a more formal way, verification shows that the program satisfies a given specification by a (mathematical) proof. Briefly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated are mostly obtained from the *system's specification*. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. Correctness is always relative to a specification, and is not an absolute property of a system. Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing).

Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model. The attractiveness of model checking comes from the fact that it is completely automatic -i.e. the learning curve for a user is very gentle- and that it offers counterexamples in case a model fails to satisfy a property serving as indispensable debugging information. Model checking requires a model of the system under consideration and a desired property and systematically checks whether or not the given model satisfies the property. Typical properties that can be checked are deadlock freedom, invariants, and request-response properties [10]. Model checking is an automated technique to check the absence of errors (i.e. property violation) and alternatively can be considered as an intelligent and effective debugging technique.

With *simulation*, a model of the system at hand is constructed and simulated. Based on inputs, execution paths of the system are examined using a simulator. A mismatch between the simulators output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice. In the other words, the main limitation of simulation is that there is no guarantee that the corner cases are ever reached (would require possibly infinite number of simulation runs, i.e. when real values, concurrency or non-determinism are in the model).

1.1.3 Testing

Testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results. In testing the implementation of the system is taken as already realized and is stimulated with certain (hopefully well-chosen) inputs and the reaction of the system is observed. Whereas verification proves conformance with a given specification, testing finds cases where a program does not meet its specification. It is important to note, that testing can never be complete, since it is built up solely of observations. Hence, only a small subset of all possible instances of a systems behavior can be taken into consideration. The purpose of testing is to make sure that a manufactured embedded system behaves as intended. Testing can be done during or after the fabrication (fabrication testing) and also after the system has been delivered to the customer (field testing).

Software testing is a dynamic technique that actually runs the system. Software testing constitutes a significant part of any software engineering project. Usually, between 30% and 50% of the total software project costs are devoted to testing [10]. Testing takes the piece of software under consideration and provides its compiled code with input, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during the test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human-beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible; in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence.

Software testing methods are traditionally divided into white- and black-box testing. White-box testing is when the tester has access to the internal data structures and algorithms including the code that implement these. Black-box testing treats the software as a “black-box” i.e. without any knowledge of internal implementation. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

Hardware testing is achievable through hardware description languages (HDL). One of the first steps in hardware design is to write down the logical structure and behavior of the circuit using an HDL which is a software-like language. This logical description is later compiled into circuit elements. Functional design verification aims to find problems at this early stage of design by analyzing the description written in HDL. The behavior and the structure of designs are usually so complex that, in many cases, the design is not entirely correct and will behave differently

than expected once implemented as a circuit. Since it is prohibitively expensive to fix problems after the design is fabricated, functional design verification proves to be indispensable by finding problems early on, before additional work is done on the design. With automatic test vector generation (see for example [26, 8, 55]), it is possible to run more tests more often and earlier in the development process which probably results in improved quality of the system. The test vector generation [25] produces a set of test vectors that include the inputs, expected outputs, and requirement traceability link. Boundary-scan, as defined by the IEEE Std.-1149.1 [94] standard, is an integrated method for testing interconnects on printed circuit boards (PCBs) that are implemented at the integrated circuit (IC) level.

Testing embedded/cyber-physical systems in their real environment may be dangerous. For example, testing control software in a nuclear power plant can be a source of serious, far-reaching problems. Model-based testing is the application of model-based design for designing and optionally executing the necessary artifacts to perform software testing. Models can be used to represent the desired behavior of the System Under Test (SUT), or to represent the desired testing strategies and testing environment. Innovative work is needed to make effective connections to the design environments and tools that can produce formal models automatically for embedded systems.

1.2 The Chess Wireless Sensor Network

The next evolutionary development step in building, utilities, industrial, home, shipboard, and transportation systems automation is represented by smart environments. Like any reactive system, the smart environment relies first and foremost on sensory data from the real world. Sensory data comes from multiple sensors of different modalities in distributed locations. The smart environment needs information about its surroundings as well as about its internal workings. Most information needed by smart environments is provided by *sensor networks*, which are responsible for sensing as well as for the first stage of the processing hierarchy. The challenges in monitoring the environment, detecting the relevant events, collecting the data, assessing and evaluating the information, formulating meaningful reports for the users, and performing decision-making and alarm functions are enormous.

A *wireless sensor network* (WSN) is a collection of nodes organized into a cooperative network. Each node has a processing capability (one or more microcontrollers, CPUs or DSP chips), may contain multiple types of memory (program, data and flash memories), has a RF transceiver (usually with a single omnidirectional antenna), has a power source (e.g., batteries and solar cells), and accommodate various sensors and actuators. The nodes communicate wirelessly and often self-organize after being deployed in an ad hoc fashion. Systems of 1000s or even 10,000 nodes are anticipated. Wireless sensor networks are currently beginning to

be deployed at an accelerated pace. Such systems can revolutionize the way we live and work. It is not unreasonable to expect that in 10-15 years that the world will be covered with wireless sensor networks which are connected to the Internet.

This new technology is exciting with unlimited potential for numerous application areas including environmental, medical, military, transportation, entertainment, crisis management, homeland defense, and smart spaces. The proper design of wireless sensor networks is challenging as it requires an broad breadth of knowledge from a wide variety of disciplines, such as communications, wireless technologies, smart sensors, self-organization and signal processing.

An effective protocol for wireless sensor networks must consume little power, avoid collisions, be implemented with a small code size and memory requirements, be efficient for a single application, and be tolerant to changing radio frequency and networking conditions. One of the greatest challenges in the design of wireless sensor networks is to find suitable mechanisms for clock synchronization.

The Chess eT International B.V. has developed a WSN platform using a gossip (epidemic) communication model. Gossiping in distributed systems refers to the repeated probabilistic exchange of information between two members. In gossip networks, information can spread within a group just as it would in real life. The main advantage of gossiping is that the absence of explicit routing provides a potentially scale free network, whereby message flooding provides robustness. In order to meet strict energy constraints, Chess used a Time Division Multiple Access (TDMA) protocol where the period in which nodes are active is limited and for the remainder of the time, nodes switch to an energy saving mode. One of the greatest challenges in the design of communication protocols is to find suitable mechanisms for clock synchronization: we must ensure that whenever some node is sending all its neighbors are listening. Each wireless sensor node has a low-cost 32 KHz crystal oscillator driving an internal clock used to determine the start and end of each slot. The TDMA time slot boundaries might drift (i.e. oscillators are sensitive to temperature changes) that makes the nodes go out of sync . Many clock synchronization protocols have been proposed for WSNs. In most of these protocols, clocks are synchronized to an accurate real-time standard like Universal Coordinated Time (UTC). However Chess has employed a different approach in which a node only needs to be synchronized to its immediate neighbors, not to faraway nodes or to UTC. In the first part of this thesis, clock synchronization in the Chess WSN is studied.

1.3 Automata Learning

The construction of models typically requires specialized expertise. It is time consuming and involves significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. In practice, 80% of software development involves legacy code, for which only poor documentation is available. Manual construction of models of legacy components is typically

very labor intensive and often not cost effective. The possible solution that is investigated in this thesis is to infer models automatically through observations and test, that is, through black-box reverse engineering.

The problem of inducing, learning or inferring grammars and automata has been studied for decades, but only in recent years grammatical inference a.k.a. grammar induction has emerged as an independent field. Grammatical inference techniques aim at building a grammar or automaton for an unknown language, given some data about this language. Within the setting of active learning, it is assumed that a learner interacts with a teacher. Inspired by the work of Angluin [6] on the L^* algorithm, Niese [76] developed an adaptation of the L^* algorithm for active learning of deterministic Mealy machines. This algorithm has been further optimized in [83]. In the algorithm it is assumed that the teacher knows a deterministic Mealy machine $\mathcal{M}$. Initially, the learner only knows the action signature (the sets of input and output symbols I and O) and her task is to learn a Mealy machine that is equivalent to $\mathcal{M}$. The teacher will answer two types of questions – output queries ("what is the output generated in response to input $i \in I^*$?") and equivalence queries ("is a hypothesized machine $\mathcal{H}$ correct, i.e., equivalent to the machine $\mathcal{M}$?"). The learner always records the current state q of Mealy machine $\mathcal{M}$. In response to query i, the current state is updated to q' and answer o is returned to the learner. At any point the learner can "reset" the teacher, that is, change the current state back to the initial state of $\mathcal{M}$. The answer to an equivalence query $\mathcal{H}$ is either yes (in case $\mathcal{M} \approx \mathcal{H}$) or no (in case $\mathcal{M} \not\approx \mathcal{H}$). Furthermore, the teacher will give the learner a counterexample that proves that the learner's hypothesis is wrong with every negative equivalence query response, that is, an input sequence $u \in I^*$ such that $obs_{\mathcal{M}}(u) \neq obs_{\mathcal{H}}(u)$. This algorithm has been implemented in the LearnLib tool [83].

State-of-the-art tools for active learning of state machines are able to learn state machines with at most in the order of 10.000 states. This is not enough for learning models of realistic software components which, due to the presence of program variables and data parameters in events, typically have much larger state spaces.

Abstraction is the key when learning behavioral models of realistic systems. Hence, in most practical applications where automata learning is used to construct models of software components, researchers manually define abstractions which, depending on the history, map a large set of concrete events to a small set of abstract events that can be handled by automata learning tools. Recently, Aarts, Jonsson & Uijen have proposed a framework for regular inference with abstraction in which, depending on the history, a large set of concrete events is mapped to a small set of abstract events [1]. Using this framework they succeeded to automatically infer models of several realistic software components with large state spaces, including fragments of the TCP and SIP protocols.

In the second part of this thesis, it is shown how such abstractions can be constructed fully automatically for a class of extended finite state machines in

which one can test for equality of data parameters, but no operations on data are allowed. This aim is reached through counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior, the abstraction is refined automatically. Using a prototype implementation of the algorithm, models of several realistic software components, including the biometric passport and the SIP protocol were learned fully automatically.

1.4 Thesis Statement

This thesis is funded by the NWO project ARTS, Abstraction Refinement for Timed Systems. In 2008, when I started as a PhD candidate, the european project Quasimodo was started which introduced several industrial challenges including Chess wireless sensor network case. In line with that project, we planned to use Counterexample-guided Abstraction Refinement (CEGAR) to verify the synchronization of a an arbitrary size WSN. Although we succeeded to use invariant proof techniques to verify an arbitrary size WSN with clique topology, and despite the fact that we discovered a potential flaw in Chess implementation, after two years of working on the project, we did not succeed to use CEGAR method in order to conquer the state space explosion problem when verifying the Chess synchronization protocol. However, in 2010, a new research project started in MBSD in the field of automata learning, which seemed to be a good place for performing CEGAR method. Within that project, we decided to design and implement a CEGAR-based algorithm which is capable of fully automatically learning a class of parametric systems. Correspondingly, this dissertation is organized in two parts.

Part one is written based on the publications

1. F. Heidarian, J. Schmaltz and F.W. Vaandrager. Analysis of a Clock Synchronization Protocol for Wireless Sensor Networks. In A. Cavalcanti and D. Dams, editors. *Proceedings FM 2009: Formal Methods, Eindhoven, The Netherlands, November 2009.* LNCS 5850, pp. 516-531, Springer-Verlag, 2009., and

2. M. Schuts, F. Zhu, F. Heidarian and F.W. Vaandrager. Modelling Clock Synchronization in the Chess gMAC WSN Protocol. In S. Andova et.al., editors. *Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09), Eindhoven, the Netherlands, 3rd November 2009.* Electronic Proceedings in Theoretical Computer Science 13, pp.41-54, 2009.

In part one, the industrial case-study of Chess on wireless sensor networks is investigated. Uppaal is used for modeling and verification of two synchronization algorithms for wireless sensor networks. Chapter 2 introduces Chess WSN, in detail. Afterwards, in chapter 3 a synchronization algorithm for wireless sensor

networks is fully described. Furthermore, the timed automata model of the synchronization protocol is depicted in detail and it is shown how Uppaal is used to extract the error scenarios presenting the situations the network goes out of sync. Based on such error scenarios, three conditions are introduced for a fully connected network to work correctly. The conditions are proved to be necessary and sufficient using invariant proof techniques. Isabelle/HOL supports the proofs. In chapter 4 another synchronization protocol, named Median, is fully described, and a detailed timed automata model of the protocol is presented in Uppaal. The model is checked for synchronization, and an error scenario showing how the network goes out of sync is presented. The error scenario is reproducible in reality. While in chapter 3 the focus of modeling is simplicity to make verification easier, in chapter 4 the model is attempted to be constructed as close to real implementation as possible. Chapter 5 concludes the research described in part one and compares the obtained results with the most related researches.

Part two is written based on the publications

1. F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata Learning Through Counterexample-Guided Abstraction Refinement. To appear in 18th International Symposium on Formal Methods (FM 2012), Paris, France, August 27-31, 2012. Springer-Verlag 2012., and

2. F. Aarts, F. Heidarian, and F.W. Vaandrager. A Theory of History Dependent Abstractions for Learning Interface Automata. April 2012. Submitted.

Part two describes how counterexample-guided abstraction refinement can be employed to expand the learning ability of automata learning tools. Chapter 6 gives a short introduction of automata learning and the way abstraction refinement can be used to strengthen the abilities of the current tools. Chapter 7 provides a solid theoretical foundation for learning interface automata using a large class of abstractions. In chapter 8, it is explained how this theoretical results support building the tool Tomte to learn a limited class of interface automata, called scalarset symbolic interface automata. Chapter 9 concludes the investigation and represents the most related works. Finally, I end the thesis in chapter 10.

Part I

Formal Analysis of Synchronization Protocols for Wireless Sensor Networks

Chapter 2

Introduction to Part One

The research reported in the first part of this thesis was carried out within the context of the EU project Quasimodo. The main goal of Quasimodo was to develop new techniques and tools for model-driven design, analysis, testing and code-generation for advanced embedded systems where ensuring quantitative bounds on resource consumption is a central problem. Case studies have been the driving momentum behind the project. Quasimodo followed an iterative approach where fundamental research on theory and algorithms —challenged by real-life case studies— was developed and implemented in methods and tools, which were evaluated through case studies. The Chess eT International B.V. was an industrial party of Quasimodo project who provided several case studies including a wireless sensor network running an epidemic communication protocol.

The first part of this thesis is devoted to modeling and analysis of synchronization algorithms for Chess wireless sensor network case study.

A *wireless sensor network* (WSN) consists of spatially distributed autonomous devices that communicate via radio and use sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations. WSNs consist of potentially thousands of nodes where each *node* comes equipped with one (or sometimes several) sensors. Each such sensor node has typically several parts: a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery or an embedded form of energy harvesting. WSNs have numerous applications, ranging from monitoring of dikes to smart kindergartens, and from forest fire detection to monitoring of the Matterhorn.

The Chess eT International B.V. develops a WSN platform using an epidemic (gossip) communication model, in the context of the MyriaNed project [80]. Figure 2.1 displays a sensor node developed by Chess.

Gossiping in distributed systems refers to the repeated probabilistic exchange of information between two members [53, 33]. The effect is that information can

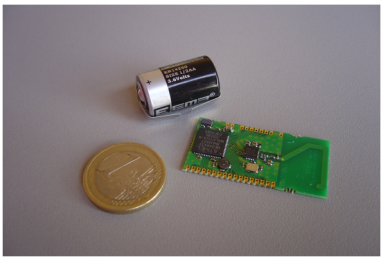

Figure 2.1: Chess MyriaNode 2.4 GHz wireless sensor node

spread within a group just as it would in real life. Their simplicity, robustness and flexibility make gossip based algorithms attractive for data dissemination and aggregation in wireless sensor networks. However, formal analysis of gossip algorithms is a challenging research problem [11]. The Chess WSN distinguishes three protocol layers: the *Medium Access Control (MAC) layer*, which is responsible for regulating the access to the wireless shared channel, the intermediate *Gossip layer*, which is responsible for insertion of new messages, forwarding of current messages and deletion of old messages, and the *Application layer*, which has the business logic that interprets messages and may generate new messages. This research concentrates on the MAC layer of the Chess WSN.

The rest of this introduction is organized as follows. Section 2.1, gives a brief account of MyriaNed design for MAC layer. Section 2.2 introduces a proposed synchronization algorithm for Chess WSN and explains how the suggested protocol is formally analyzed in chapter 3 this thesis. Section 2.3 presents the basics of Median algorithm which is the focus of chapter **??**.

2.1 Chess MAC model

The MAC layer uses specific protocols to ensure that signals sent from different stations across the same channel don't collide, as RF broadcasting is used to transfer message. Characteristics of the other layers influence the design decisions for the MAC layer. For instance, the redundant nature of the Gossip layer justifies occasional message loss in the MAC layer.

Chess used a Time Division Multiple Access (TDMA) protocol for the MAC layer. In this approach, time is divided in fixed length *frames*, and each frame is subdivided into *slots* (see Figure 2.2). Slots can be either *active* or *idle*. During active slots, a node is either listening for incoming messages from neighboring nodes ("*RX*") or it is sending a message itself ("*TX*"). During idle slots a node is switched to energy saving mode. In WSNs, nodes are usually battery operated devices with an expected uninterrupted field deployment of several years. Hence, energy efficiency is a major concern in the design of WSNs. For this reason, in

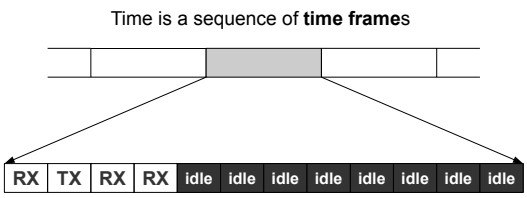

Figure 2.2: The structure of a time frame

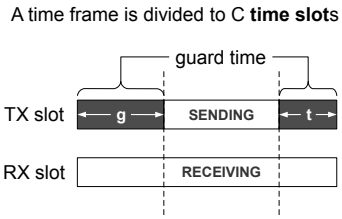

Figure 2.3: TX and RX slots

MyriaNed design the number of active slots is typically much smaller than the total number of slots (less than 1% in the current implementation [80]). The active slots are placed in one contiguous sequence which is placed at the beginning of the frame. A node can only transmit a message once per time frame in its TX slot. If two neighboring nodes choose the same send slot, a collision will occur in the intersection of their ranges preventing delivery of either node's message in that intersection. Ideally, no neighboring pair would ever choose the same send slot. This has proven to be very hard to achieve, especially in settings with node mobility. In this thesis, the issue of slot allocation is not addressed and it is simply assumed that the TX slots of all nodes are fixed and have been chosen in such a way that no collisions occur. Actually, receiving is typically more expensive than sending, as radio needs to be turned on longer. Furthermore, receiving usually consumes more energy per bit.

One of the greatest challenges in the design of the MAC layer is to find suitable mechanisms for clock synchronization, that is a distributed algorithm to ensure that the start of all active periods of the nodes are synchronous. More precisely, it must be guaranteed that whenever some node is sending all its neighbors are listening.

In the setting of Chess, each wireless sensor node comes equipped with a low-cost 32 KHz crystal oscillator that drives an internal clock that is used to determine the start and end of each slot. This may cause the TDMA time slot boundaries to drift and thus lead to situations in which nodes get out of sync. To overcome this problem, the notion of *guard time* is introduced: at the beginning of its TX slot, before actually starting transmission, a sender waits a certain amount of time for the receiver to be ready to receive messages. Similarly, the sender also waits for some time period at the end of its TX slot (see Figure 2.3).

In the implementation of Chess, each slot consisted of 29 clock cycles, out of which 18 cycles were used as guard time. Assegei [7] calculated how the battery life of a wireless sensor node is influenced by the guard time. Figure 2.4, taken from [7], summarizes these results. Clearly, it is of vital importance to reduce the guard time as much as possible, since this directly affects the battery life, which is a key characteristics of WSNs. Intuitively, larger guard-time leads to larger slots which leads to larger active time which implies more energy consumption. Reduction of

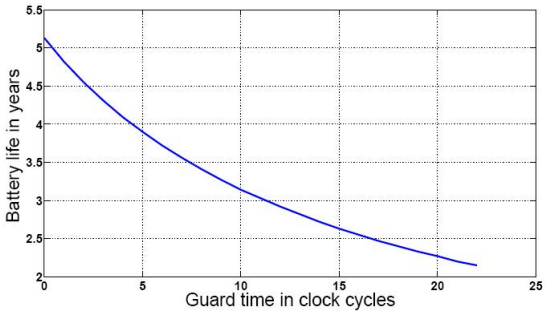

Figure 2.4: Battery life as a function of guard time

the guard time is possible if the hardware clocks are properly synchronized.

To experiment with its designs, Chess builds prototypes and uses advanced simulation tools. However, due to the huge number of possible network topologies and clock speeds of nodes, although it is not difficult to discover flaws in the clock synchronization algorithm via these methods, it is difficult to understand the root cause of the flaws and to provide objective evidence that the algorithm is correct under all circumstances.

Timed automata model checking has been successfully used for the analysis of worst case scenarios for protocols that involve clock synchronization, see for instance [16, 41, 97]. To enable model checking, models need to be much more abstract than for simulation, and also the size of networks that can be tackled is much smaller, but the big advantage is that the full state space of the model can be explored.

The purpose of the research of this thesis was to use timed automata model checking to help Chess with designing synchronization protocols for their WSN.

2.2 MyriaNed Protocol

In chapter 3, based on the model of Chess for the MAC layer, a synchronization algorithm is proposed for WSN in which a node adjusts its clock whenever a message arrives, and a timed automata model is presented for the suggested algorithm. Then the use of timed automata model checker Uppaal [15] for analyzing WSNs with full connectivity is explained. Various instances are verified and three different scenarios are identified which may lead to situations where the network is out of sync. Besides, a full parametric analysis of the protocol for cliques (networks with a connection between every pair of nodes), is presented, that is, constraints on the parameters are given that are both necessary and sufficient for correctness. The results are checked using the proof assistant Isabelle [77]. In order to make verification feasible, the model of chapter 3 abstracts from several aspects in the implementation, including radio switching time: there is some time involved in

the transition from sending mode to receiving mode (and vice versa), which in some cases may affect the correctness of the algorithm. Finally, a result for the special case of line topologies is presented: for any instantiation of the parameters, the protocol will eventually fail if the network grows. Although this approach has advantages, the practical usefulness of this algorithm still needs to be explored further.

2.3 Median Protocol

Chess has implemented *Median* algorithm, an extension of an algorithm proposed by Tjoa et al [91], on WSN. The idea is that in every frame, each node computes its phase error to any of its direct neighbors. After the last active slot, each node adjusts its frame length by the median of the phase errors of its immediate neighbors. In chapter 4, a detailed model of the Chess Median algorithm is presented using the input language of Uppaal. The aim is to construct a model that comes as close as possible to the specification of the clock synchronization algorithm presented in [80]. Nevertheless, the model still does not incorporate some features of the full algorithm and network, such as dynamic slot allocation, synchronization messages, uncertain communication delays, and unreliable radio communication. At places where the informal specification of [80] was incomplete or ambiguous, the engineers from Chess kindly provided additional information on the way these issues are resolved in the current implementation of the network [100]. The Median algorithm works reasonably well in practice, but by means of simulation experiments, Assegei [7] points out that the performance of the Median algorithm decreases if the network becomes more dynamic. In some test cases where new nodes join or networks merge, the algorithm fails to converge or nodes may stay out of sync for a certain period of time. Analysis with Uppaal as presented in chapter 4, confirms these results. In fact, it is shown in chapter 4 that the situation is even worse: in certain cases a static, fully synchronized network may eventually become unsynchronized if the Median algorithm is used, even in a setting with infinitesimal clock drifts. This theoretical result has been later reproduced experimentally in a real network of Chess. Assegei [7] proposes a variation of the Median algorithm that uses Kalman filters, but as it is shown in [42], also this variation leads to serious synchronization problems.

Chapter 3

Modeling and Verification of MyriaNed Synchronization Protocol for Wireless Sensor Networks

In this chapter, the MyriaNed algorithm for the Chess WSN is analyzed. In this protocol a node adjusts its clock whenever a message arrives. This chapter is structured as follows. In Section 3.1, the synchronization algorithm is modeled using timed automata. Section 3.2 describes the use of the timed automata model checker Uppaal to analyze WSNs with full connectivity. Various instances are verified and three different scenarios that may lead to situations where the network is out of sync, are identified. Section 3.3 presents a full parametric analysis of the protocol for cliques. In Section 3.4 an exhaustive analysis is reported using Uppaal of all networks with 4 nodes. Section 3.5 presents a result for the special case of line topologies: for any instantiation of the parameters, the protocol will eventually fail if the network grows. Section 3.6, finally, discusses related work and draws conclusions.

Uppaal models, Isabelle sources and invariant proofs for this study are available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/HSV09/`.

3.1 Uppaal Model

In this section, the Uppaal model of the Chess protocol is described. A detailed account of the timed automata model checking tool Uppaal, is presented in [15, 14] and the website `http://www.uppaal.com`.

A wireless sensor network is defined as a finite, fixed set of wireless nodes

$\mathsf{Nodes} = \{0, \ldots, \mathsf{N}-1\}$. The behavior of each individual node $i \in \mathsf{Nodes}$ is described by three timed automata: **Clock**(i), **WSN**(i) and **Synchronizer**(i). Automaton **Clock**(i) models the hardware clock of the node, automaton **WSN**(i) takes care of sending messages, and the **Synchronizer**(i) automaton resynchronizes the hardware clock upon receipt of a message. The complete model consists of the composition of timed automata **Clock**(i), **WSN**(i) and **Synchronizer**(i), for each $i \in \mathsf{Nodes}$.

Figure 3.1 illustrates the architecture of the model for a single node i. For each

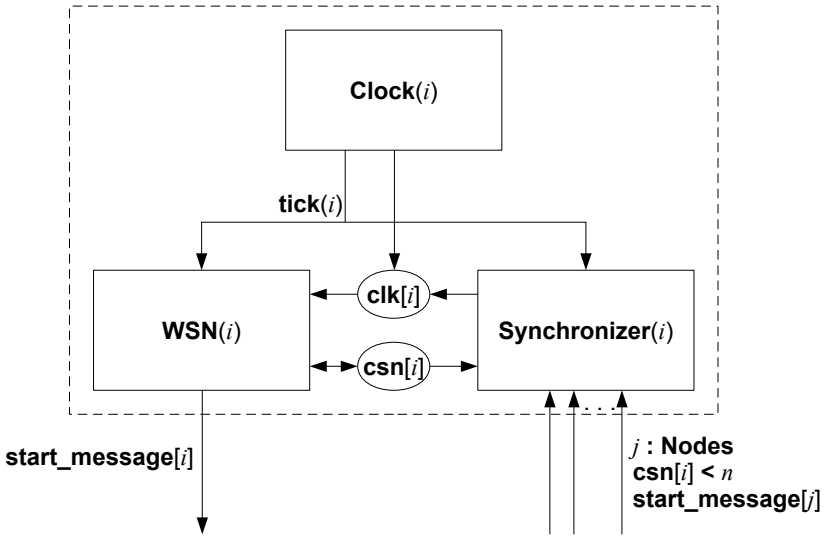

Figure 3.1: Model architecture for node i

i, there is a state variable $\mathsf{clk}[i]$ that records the (integer) value of i's hardware clock (initially 0), and a variable $\mathsf{csn}[i]$ that records the current slot number of node i (also 0 initially). Variable $\mathsf{clk}[i]$ is incremented cyclically whenever **Clock**$[i]$ ticks, but it can also be reset by **Synchronizer**$[i]$. Automaton **WSN**$[i]$ reads $\mathsf{clk}[i]$ in order to determine when to transmit. Automaton **WSN**$[i]$ both reads and write variable $\mathsf{csn}[i]$. The **Synchronizer**$[i]$ needs to read variable $\mathsf{csn}[i]$ in order to determine whether node i is active or idle. In UPPAAL, a broadcast channel can be used to synchronize transitions of multiple automata. If a is a broadcast channel and one automaton in the network is in a state with an outgoing $a!$ transition, then this transition may always occur (provided the guard evaluates to true). In this case, the transition synchronizes with the $a?$ transitions of all automata that enable such a transition. Automata that do not enable an $a?$ transition remain in the same state. Within the model, broadcast channel $\mathsf{tick}[i]$ is used to synchronize the activities within node i, and broadcast channel $\mathsf{start\_message}[i]$ is used to inform all the nodes in the network that node i has started transmission. More

specifically, automaton **WSN**$[i]$ performs a $\mathsf{start\_message}[i]!$ action to indicate that node i starts transmission, and whenever some node j starts transmission and node i is in an active slot ($\mathsf{csn}[i] < n$), automaton **Synchronizer**$[i]$ may perform a $\mathsf{start\_message}[j]?$ transition.

Table 4.1 lists the parameters (constants in UPPAAL terminology) that are used in the model, together with some basic constraints. The domain of all parameters is the set of natural numbers.

Parameter	Description	Constraints
N	number of nodes	$1 < \mathsf{N}$
C	number of slots in a time frame	$0 < \mathsf{C}$
n	number of active slots in a time frame	$0 < \mathsf{n} \leq \mathsf{C}$
$\mathsf{tsn}[i]$	TX slot number for node $i \in \mathsf{Nodes}$	$0 \leq \mathsf{tsn}[i] < \mathsf{n}$
$\mathsf{k_0}$	number of clock ticks in a time slot	$0 < \mathsf{k_0}$
g	guard time	$0 < \mathsf{g}$
t	tail time	$0 < \mathsf{t}$, $\mathsf{g} + \mathsf{t} + 2 \leq \mathsf{k_0}$
min	minimal time between two clock ticks	$0 < \mathsf{min}$
max	maximal time between two clock ticks	$\mathsf{min} \leq \mathsf{max}$

Table 3.1: Protocol parameters

3.1.1 Clock

Timed automaton **Clock**(i), displayed in Figure 3.2, models the behavior of the hardware clock of node i. It has a single location and a single transition. It comes equipped with a local clock variable x, which is initially 0, that is used to measure the time in between clock ticks. Whenever x reaches the value min, the automaton enables a $\mathsf{tick}[i]!$ action. The $\mathsf{tick}[i]!$ action must occur before x has reached value max. Then x is reset to 0 and the (integer) value of i's hardware clock $\mathsf{clk}[i]$ is incremented by 1. For convenience and in order to make model checking feasible, the hardware clock is reset after $\mathsf{k_0}$ ticks, that is, the clock takes integer values modulo $\mathsf{k_0}$ (UPPAAL's modulo operator % is used). This is not an essential modeling assumption and it is easy to change it. In the implementation of the protocol, the clock domain is larger and additional program variables are used to record, for instance, the clock value at which a frame starts. However, in order to avoid state space explosion, it is tried to reduce the number of state variables and the domains of these variables.

3.1.2 Wireless Sensor Node

Automaton **WSN**(i), displayed in Figure 3.3, is the most important component in the model. It has three locations and four transitions. The automaton stays

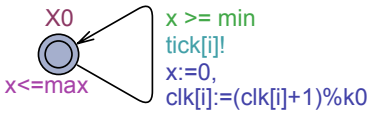

Figure 3.2: Timed automaton **Clock**(i)

in initial location WAIT until the current slot number of i equals the TX slot number of i ($\mathsf{csn}[i] = \mathsf{tsn}[i]$) and the g^{th} clock tick in this slot occurs. It then jumps to location GO_SEND. This is an urgent location that is left immediately via a start_message$[i]$!-transition to location SENDING. Broadcast channel start_message$[i]$ is used to inform all neigboring nodes that a new message transmission has started. The automaton stays in location SENDING until the start of the tail interval, that is, until the $(\mathsf{k_0} - \mathsf{t})^{th}$ tick in the current slot (cf. Figure 2.3), and then returns to location WAIT. At the end of each slot, that is, when the $\mathsf{k_0}^{th}$ tick occurs, the automaton increments its current slot number (modulo C).

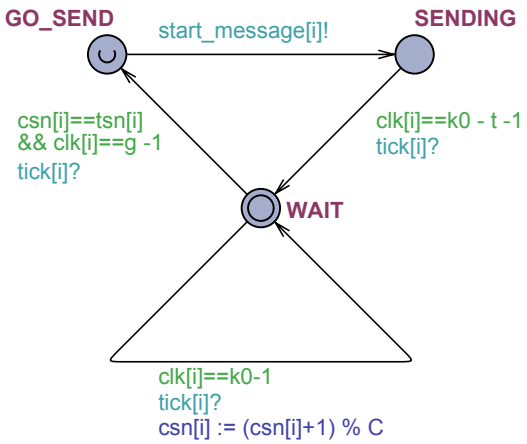

Figure 3.3: Timed automaton **WSN**(i)

3.1.3 Synchronizer

Automaton **Synchronizer**(i), displayed in Figure 3.4, is the last component of the model. It performs the role of the clock synchronizer in the TDMA protocol. The automaton has two locations and two transitions. The automaton waits in its initial location S0 until it detects the start of a new message, that is, until a start_message$[j]$? event occurs, for some j. The UPPAAL select statement is used

to nondeterministically select a $j \in \mathsf{Nodes}$. The automaton then moves to location S1, provided node i is active ($\mathsf{csn}[i] < \mathsf{n}$). Remember that at the moment when the start_message[j]? event occurs, the hardware clock of node j, clk[j], has value g. Therefore, node i resets its own hardware clock clk[i] to $\mathsf{g}+1$ upon occurrence of the first clock tick following the start_message[j]? event. The automaton then returns to its initial location S0.

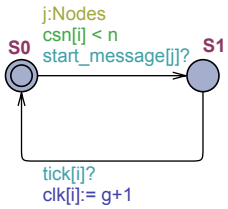

Figure 3.4: Timed automaton **Synchronizer**(i)

In the model there is no delay between sending and receipt of messages. Following Meier & Thiele [70], it is assumed delay uncertainties are negligible, and therefore the delays are eliminated from the analysis. When communication is infrequent, this is reasonable since the impact of clock drift dominates over the influence of delay uncertainties.

Automaton **Synchronizer**(i) has no constraint on the value of j, that is, node i is assumed to be able to receive messages from any node in the network. Hence the network has full connectivity. It is easy to generalize this model to a setting with arbitrary network topologies by adding a guard $\mathsf{neighbor}(i,j)$ to the transition from S0 to S1 that indicates that i is a direct neighbor of j. It is assumed that $\mathsf{neighbor}(i,j) \Rightarrow i \neq j$. The $\mathsf{neighbor}(i,j)$ predicate does not have to be symmetric since in a wireless sensor network it may occur that i can receive messages from j, but not vice versa. For networks with full connectivity, it is assumed that all nodes have unique TX slot numbers:

$$i \neq j \quad \Rightarrow \quad \mathsf{tsn}[i] \neq \mathsf{tsn}[j] \tag{3.1}$$

For networks that are not fully connected, this assumption generalizes to the requirements that neighboring nodes have distinct TX slot numbers, and distinct nodes with the same TX slot number do not have a common neighbor:

$$\mathsf{neighbor}(i,j) \quad \Rightarrow \quad \mathsf{tsn}[i] \neq \mathsf{tsn}[j] \tag{3.2}$$

$$\mathsf{neighbor}(i,j) \wedge \mathsf{neighbor}(i,k) \quad \Rightarrow \quad \mathsf{tsn}[j] \neq \mathsf{tsn}[k] \tag{3.3}$$

3.2 Uppaal Analysis Results for Cliques

A wireless sensor network is called *synchronized* if whenever a node is sending, all neighboring nodes have the same slot number as the sending node. For networks

with full connectivity this means that all nodes in the network agree on the current slot, which leads to the following formal definition of correctness.

Definition A network with full connectivity is *synchronized* if and only if for all reachable states $(\forall i, j \in \mathsf{Nodes})(\mathsf{SENDING}_i \Rightarrow \mathsf{csn}[i] = \mathsf{csn}[j])$.

The objective is to find necessary and sufficient constraints on the system parameters that ensure that a network with full connectivity is synchronized. To this end, different values are assigned to the parameters of the model and UPPAAL is used to verify the property of Definition 3.2. Based on the outcomes (and in particular the counterexamples generated by UPPAAL) general constraints are derived. For networks with up to 4 nodes, the UPPAAL model checker is able to explore the state space within a few seconds.

Table 3.2 shows some example values of the parameters for which the model is synchronized. In fact, via a series of model checking experiments, using binary search, min and max are found to be the smallest consecutive natural numbers for which the model with the values assigned to N, C, n, k_0 and g is synchronized. Note that for correctness of the WSN algorithm the exact values of min and max are not important: what matters is their ratio. By setting $\mathsf{min} = m$, $\mathsf{max} = m+1$ and letting m grow, the hardware becomes more and more accurate, until (hopefully) we reach the point at which the algorithm becomes correct. Parameter t is chosen equal to g and $\mathsf{tsn}(i)$ is chosen equal to i. n, k_0 and g are kept constant while C, the number of slots in a frame, is varying. Observe that if the value of C increases also the values of min and max increase, i.e., if the length of a frame increases then the hardware clocks must become more accurate to maintain synchronization.

N	2			3			4		
C	6	8	10	6	8	10	6	8	10
n	4	4	4	4	4	4	4	4	4
k_0	10	10	10	10	10	10	10	10	10
g	2	2	2	2	2	2	2	2	2
min	49	69	89	39	59	79	29	49	69
max	50	70	90	40	60	80	30	50	70

Table 3.2: Some UPPAAL verification results

Observe that these parameter values are not realistic: a realistic clock accuracy is around 30 ppm (parts-per-million), C is about 1000 (instead of 10), and g is 9 (instead of 2). UPPAAL cannot handle realistic values because of the state explosion problem. Nevertheless, as we will see, the counterexamples provided by UPPAAL do provide insight.

In Table 3.3, all the parameters are kept constant and then the values of min and max are considered for different numbers of nodes when n changes. Since, in

accordance with the specification of the protocol, only slot allocations in which the sending slots are placed at the very beginning of a frame are considered, increasing n has no impact on network behavior: when no node is transmitting anyway, it makes no difference whether nodes are sleeping or listening. In Table 3.4, all the

N	2				3				4			
C	20	20	20	20	20	20	20	20	20	20	20	20
n	4	5	10	15	4	5	10	15	4	5	10	15
k_0	10	10	10	10	10	10	10	10	10	10	10	10
g	2	2	2	2	2	2	2	2	2	2	2	2
min	189	189	189	189	179	179	179	179	169	169	169	169
max	190	190	190	190	180	180	180	180	170	170	170	170

Table 3.3: Numerical results, changing n

parameters are kept constant and then the smallest values of min and max are considered for different number of nodes when k_0 changes. It turns out that increasing k_0 necessitates the increase of min and max. In Table 3.5, all the parameters are

N	2				3				4			
C	6	6	6	6	6	6	6	6	6	6	6	6
n	4	4	4	4	4	4	4	4	4	4	4	4
k_0	5	10	15	20	5	10	15	20	5	10	15	20
g	2	2	2	2	2	2	2	2	2	2	2	2
min	24	49	74	99	19	39	59	79	14	29	44	79
max	25	50	75	100	20	40	60	80	15	30	45	80

Table 3.4: Numerical results, changing k_0

kept constant and then the smallest values of min and max are considered for different number of nodes when g changes. Increasing g, facilitates the decrease of min and max.

The concrete counterexamples produced by the UPPAAL model checker can easily be transformed into parametric counterexamples: it is enough to replace the concrete values of the timing constants by parameters and collect the constraints on these parameters that are imposed by the guards and invariants that occur in the counterexample execution. Inspection of the counterexamples for the WSN protocol, which one can rerun step by step in the simulator, reveals that there are essentially three different scenarios that may lead to a state in which the network is not synchronized. In order to describe these scenarios parametrically at an abstract level, a bit of notation is needed. $s \in \{0, \ldots, \mathsf{C} - 1\}$ is said to be a

N	2			3			4		
C	6	6	6	6	6	6	6	6	6
n	4	4	4	4	4	4	4	4	4
k_0	10	10	10	10	10	10	10	10	10
g	2	3	4	2	3	4	2	3	4
min	49	24	16	39	19	13	29	14	9
max	50	25	17	40	20	14	30	15	10

Table 3.5: Numerical results, changing g

transmitting slot, notation $\mathsf{TX}(s)$, if there is some node i that is transmitting in s, that is,

$$\mathsf{TX}(s) \quad \Leftrightarrow \quad (\exists i \in \mathsf{Nodes})(\mathsf{tsn}[i] = s).$$

$\mathsf{PREV}(s)$ denotes the nearest transmitting slot that precedes s (cyclically). Formally, function $\mathsf{PREV} : \{0, \ldots, \mathsf{C}-1\} \to \{0, \ldots, \mathsf{C}-1\}$ is defined by

$$\mathsf{PREV}((s+1)\%\mathsf{C}) \;=\; \begin{cases} s & \text{if } \mathsf{TX}(s) \\ \mathsf{PREV}(s) & \text{otherwise} \end{cases} \tag{3.4}$$

$\mathsf{D}(s)$ denotes the number of slots visited when going from $\mathsf{PREV}(s)$ to s, that is, $\mathsf{D}(s) = (s - \mathsf{PREV}(s))\%C$. $\mathsf{M} = \max_s \mathsf{D}(s)$ is defined to be the maximal distance between transmitting slots. As we will see, M plays a key role in defining correctness.

3.2.1 Scenario 1: Fast Sender - Slow Receiver

In the first error scenario, a sending node is proceeding maximally fast whereas a receiving node runs maximally slow. The sender starts with the transmission of a message while the receiver is still in an earlier slot. The scenario is illustrated in Figure 3.5. It starts when the fast and the slow node receive a synchronization message. Immediately following receipt of this message (at the same point in time), the hardware clock of fast node ticks and the synchronizer resets this clock to $\mathsf{g}+1$. Now, in the worst case, it may take $\mathsf{M} \cdot \mathsf{k_0} - 1$ ticks before the fast node is in its TX slot with its hardware clock equal to g. Since the hardware clock of the fast node ticks maximally fast, the length of the corresponding time interval is $(\mathsf{M} \cdot \mathsf{k_0} - 1) \cdot \mathsf{min}$. The slow node will reach the TX slot of the fast node after $\mathsf{M} \cdot \mathsf{k_0} - \mathsf{g}$ ticks. With a clock that ticks maximally slow, this may take $(\mathsf{M} \cdot \mathsf{k_0} - \mathsf{g}) \cdot \mathsf{max}$ time. If $(\mathcal{M} \cdot \mathsf{k_0} - \mathsf{g}) \cdot \mathsf{max}$ is greater than or equal to $(\mathsf{M} \cdot \mathsf{k_0} - 1) \cdot \mathsf{min}$ then we may end up in a state where the network is no longer synchronized since the fast node is sending before the slow node has moved to the same slot. Hence, in order to exclude this scenario, we must have:

$$(\mathsf{M} \cdot \mathsf{k_0} - \mathsf{g}) \cdot \mathsf{max} \quad < \quad (\mathsf{M} \cdot \mathsf{k_0} - 1) \cdot \mathsf{min} \tag{3.5}$$

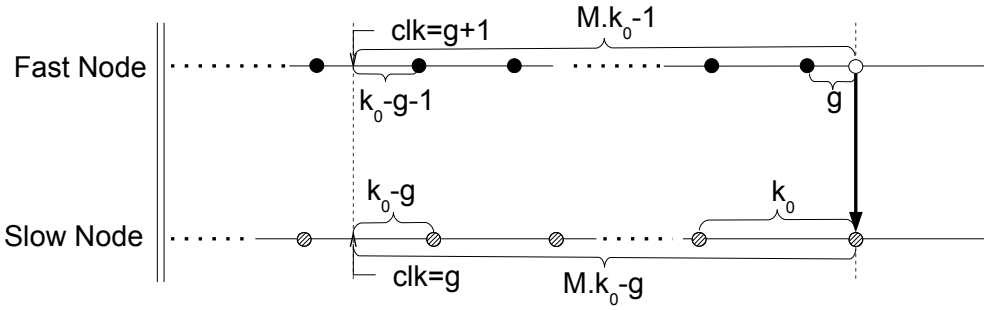

Figure 3.5: Scenario 1: Fast Sender - Slow Receiver

This constraint is consistent with the results in Table 3.2. Consider, for instance the first column. According to UPPAAL the protocol is correct if $\mathsf{N} = 2$, $\mathsf{C} = 6$, $\mathsf{n} = 4$, $\mathsf{k_0} = 10$, $\mathsf{g} = 2$, $\mathsf{min} = 49$ and $\mathsf{max} = 50$. Since it is assumed that the two nodes are sending in the first two slots of a frame, it is easy to see that $\mathsf{M} = 5$. Now we can verify that

$$(5 \cdot 10 - 2) \cdot 50 \;=\; 48 \cdot 50 \;=\; 2400 \;<\; 2401 \;=\; 49 \cdot 49 \;=\; (5 \cdot 10 - 1) \cdot 49$$

However, if we increase the clock drift slightly by setting min and max to 48 and 49, respectively, then the protocol fails according to UPPAAL. And indeed

$$(5 \cdot 10 - 2) \cdot 49 \;=\; 48 \cdot 49 \;=\; 2352 \;=\; 49 \cdot 48 \;=\; (5 \cdot 10 - 1) \cdot 48$$

Instead of the lower bound min and the upper bound max on the time between clock ticks, sometimes it is convenient to consider the ratio

$$\rho = \frac{\mathsf{min}}{\mathsf{max}}$$

Since $0 < \mathsf{min} \leq \mathsf{max}$, it follows that ρ is contained in the interval $(0, 1]$. The following elementary lemma turns out to be quite useful.

Lemma 3.2.1 *Constraint (3.5) is equivalent to* $\mathsf{g} > (1 - \rho) \cdot \mathsf{M} \cdot \mathsf{k_0} + \rho$.

This implies that the worst case scenario occurs when the distance between TX slots is maximal: if the constraint holds for M it also holds when we replace M by a smaller value.

The Chess implementation Constraint (3.5) allows us to infer a lower bound on the guard time g. In the current implementation of the protocol by Chess [80], a quartz crystal oscillator is used with a clock drift rate θ of at most 20 ppm. This means that

$$\rho \;=\; \frac{1-\theta}{1+\theta} \;=\; \frac{1 - 20 \cdot 10^{-6}}{1 + 20 \cdot 10^{-6}} \approx 0,99996$$

In the Chess implementation, one time frame lasts for about 1 second. It consists of $\mathsf{C} = 1129$ slots and each slot consists of $\mathsf{k_0} = 29$ clock ticks. The number of active slots is small ($\mathsf{n} = 10$). A typical value for M is $\mathsf{C} - \mathsf{n} = 1119$. Hence

$$\mathsf{g} > (1 - \rho) \cdot \mathsf{M} \cdot \mathsf{k_0} + \rho \approx 0,00004 \cdot 1119 \cdot 29 + 0,99996 \;=\; 2.298$$

Thus, according to the theoretical model, a value of $\mathsf{g} = 3$ should suffice. Chess actually uses a guard time of 9. Of course one should realize here that the model is overly simplified and, for instance, does not take into account (uncertainty in) message delays and partial connectivity. We will see that these restrictions greatly influence the minimal guard time.

3.2.2 Scenario 2: Fast Receiver - Slow Sender - before transmission

In the second error scenario, a receiving node runs maximally fast whereas a sending node proceeds maximally slow. The receiving node already leaves the slot in which it should receive a message from the sender before the sender has even started transmission. This scenario is illustrated in Figure 3.6. Again, the scenario

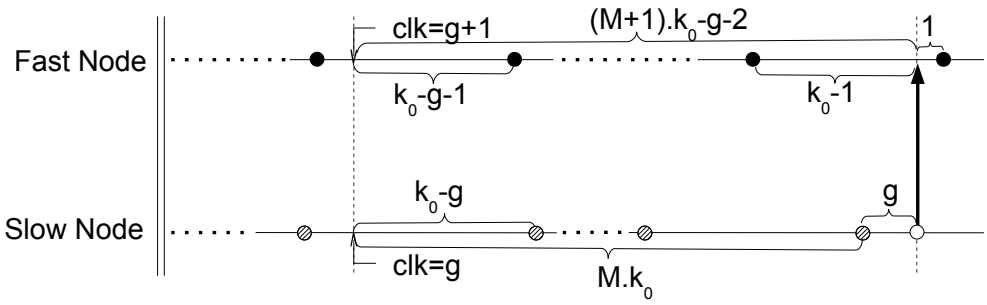

Figure 3.6: Scenario 2: Fast Receiver - Slow Sender - before transmission

starts when the fast and the slow node receive a synchronization message. But now the node that has to send the next message runs maximally slow. It sends this message after $\mathsf{M} \cdot \mathsf{k_0}$ ticks have occurred, which takes $\mathsf{M} \cdot \mathsf{k_0} \cdot \mathsf{max}$ time. Meanwhile, the fast node has made maximal progress: immediately after receipt of the first synchronization message (at the same point in time), the hardware clock of the fast node ticks and the synchronizer resets this clock to $\mathsf{g} + 1$. Already after $(\mathsf{k_0} - \mathsf{g} - 1) \cdot \mathsf{min}$ time the node proceeds to the next slot. Another $(\mathsf{M} \cdot \mathsf{k_0} - 1) \cdot \mathsf{min}$ time units later the fast node sets its clock to $\mathsf{k_0} - 1$ and is about to leave the slot in which the slow node will send a message. If the slow node starts transmission after this point it is too late: after the next clock tick the fast node will increment its slot counter and the network is no longer synchronized. In order to exclude the

second scenario, the following constraint must hold:

$$\mathsf{M} \cdot \mathsf{k_0} \cdot \mathsf{max} \quad < \quad ((\mathsf{M}+1) \cdot \mathsf{k_0} - \mathsf{g} - 2) \cdot \mathsf{min} \tag{3.6}$$

Also this constraint can be rewritten:

Lemma 3.2.2 *Constraint (3.6) is equivalent to* $\mathsf{g} < (1 - \frac{1}{\rho}) \cdot \mathsf{M} \cdot \mathsf{k_0} + \mathsf{k_0} - 2$.

Thus constraint (3.6) imposes an upper bound on guard time g. Since in practice one will always try to minimize the guard time in order to save energy, this constraint is only of theoretical interest. If we fill in the values of Example 3.2.1, we obtain $\mathsf{g} < 25.8$, which is close to the slot length $\mathsf{k_0} = 29$.

3.2.3 Scenario 3: Fast Receiver - Slow Sender - during transmission

The third scenario involves a fast receiver and a slow sender. The receiver moves to a new slot while the sender is still transmitting a message. Figure 3.7 illustrates the scenario. As in the previous scenarios, the hardware clock of the fast node is set to $\mathsf{g}+1$ immediately after receipt of the synchronization message.

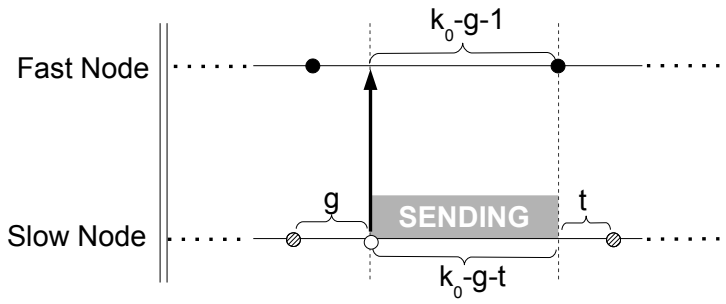

Figure 3.7: Scenario 3:Fast Receiver- Slow Sender - during transmission

To exclude this scenario, the following condition should be satisfied:

$$(\mathsf{k_0} - \mathsf{g} - \mathsf{t}) \cdot \mathsf{max} \quad < \quad (\mathsf{k_0} - \mathsf{g} - 1) \cdot \mathsf{min} \tag{3.7}$$

Essentially, constraint (3.7) provides a lower bound on t: to rule out the scenario in Figure 3.7, the sender should wait long enough before proceeding to the next slot.

Lemma 3.2.3 *Constraint (3.7) is equivalent to* $\mathsf{t} > (1-\rho)(\mathsf{k_0} - \mathsf{g}) + \rho$.

If we fill in the values of Example 3.2.1 with g set to 3, we obtain $\mathsf{t} > 1.001$. Hence a value of $\mathsf{t} = 2$ should suffice. Hence, for the simple case of a static network with full connectivity and no uncertainty in message delays, we only need to reserve 5 clock cycles for guard and tail time together. In Section 3.5, we will see that for different network topologies indeed much larger values are required.

3.3 Proving Sufficiency of the Constraints

This section outlines the proof that the three constrains derived in Section 3.2 are sufficient to ensure synchronization in networks with full connectivity.

First the key invariants used in the proof are presented and then the formalization of the full proof using Isabelle/HOL is discussed.

3.3.1 Invariants

Let's start the proof by stating some elementary invariants.

Lemma 3.3.1 *For any network with full connectivity the following invariant assertions hold, for all reachable states and for all* $i \in \mathsf{Nodes}$*:*

$$0 \leq \mathsf{x}_i \leq \mathsf{max} \tag{3.8}$$

$$0 \leq \mathsf{clk}[i] < \mathsf{k_0} \tag{3.9}$$

$$0 \leq \mathsf{csn}[i] < \mathsf{C} \tag{3.10}$$

$$\mathsf{GO\_SEND}_i \Rightarrow \mathsf{x}_i = 0 \tag{3.11}$$

$$\mathsf{GO\_SEND}_i \Rightarrow \mathsf{csn}[i] = \mathsf{tsn}[i] \tag{3.12}$$

$$\mathsf{GO\_SEND}_i \Rightarrow \mathsf{clk}[i] \in \{\mathsf{g}, \mathsf{g}+1\} \tag{3.13}$$

$$\mathsf{SENDING}_i \Rightarrow \mathsf{csn}[i] = \mathsf{tsn}]i] \tag{3.14}$$

$$\mathsf{SENDING}_i \Rightarrow \mathsf{g} \leq \mathsf{clk}[i] < \mathsf{k_0} - \mathsf{t} \tag{3.15}$$

Invariants (3.8)-(3.10) assert that the state variables indeed take values in their intended domains: clock variables stay within the (real-valued) range $[0, \mathsf{max}]$, hardware clocks stay within the integer range $[0, \mathsf{k_0})$, and current slot numbers stay within the integer range $[0, \mathsf{C})$. Invariants (3.11)-(3.15) directly follow from the definitions of the individual automata in the network. For invariant (3.13), observe that since the tick?-transition from WAIT to GO_SEND may synchronize with the tick?-transition from S1 to S0, the value of $\mathsf{clk}[i]$ in $\mathsf{GO\_SEND}_i$ is potentially $\mathsf{g}+1$.

In order to be able to state more interesting invariants, two auxiliary global history (or ghost) variables are introduced. Clock y records the time that has elapsed since the last synchronization message (or the beginning of the protocol). Variable last records the last slot in which a synchronization message has been sent (initially $\mathsf{last} = -1$). Figure 3.8 shows the version of the $\mathbf{WSN}(i)$ automaton obtained after adding these variables.

The only change is that upon occurrence of a synchronization $\mathsf{start\_message}[i]!$ clock y is reset to 0 and variable last is reset to $\mathsf{csn}[i]$. First a few basic invariants are stated which restrict the values of the new variables.

Lemma 3.3.2 *For any network with full connectivity the following invariant as-*

sertions hold, for all reachable states and for all $i \in \mathsf{Nodes}$*:*

$$0 \leq \mathsf{y} \tag{3.16}$$

$$-1 \leq \mathsf{last} < \mathsf{C} \tag{3.17}$$

$$\mathsf{S1}_i \Rightarrow \mathsf{y} \leq \mathsf{x}_i \tag{3.18}$$

$$\mathsf{last} = -1 \Rightarrow \mathsf{S0}_i \tag{3.19}$$

Invariant (3.16) says that y is always nonnegative, and invariant (3.17) says that last takes values in the integer domain $[-1, \mathsf{C} - 1)$. If the system is in $\mathsf{S1}_i$ then a synchronization occurred after the last clock tick (invariant (3.18)), and if the system is in $\mathsf{S0}_i$ then no synchronization occurred yet (invariant (3.19)).

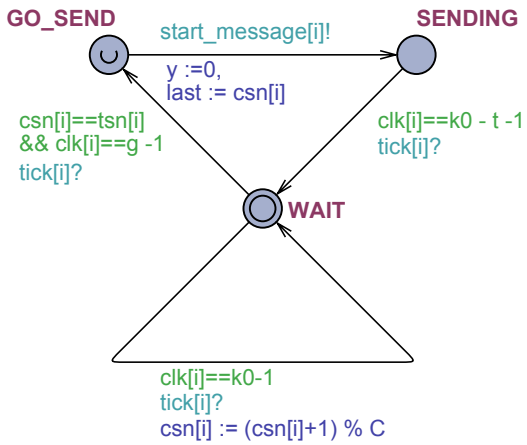

Figure 3.8: **WSN**(i) with history variables

The key idea behind the correctness proof is that, given the local state of some node i and the value of last, we can compute the number $c(i)$ of ticks of i's hardware clock that has occurred since the last synchronization. Since the minimal and maximal clock speeds are known, we can then derive an interval that contains the value of y, the amount of real-time that has elapsed since the last synchronization. Next, given the value of y, we can compute an interval that contains the value of $c(j)$, for arbitrary node j. Once we know the value of $c(j)$, this gives us some information about the local state of node j. Through these correspondences, we are able to infer that if node i is sending the slot number of i and j must be equal.

Formally, for $i \in \mathsf{Nodes}$, the state function $c(i)$ is defined by

$$
\begin{array}{rcl}
c(i) & = & \textbf{if } \mathsf{last} = -1 \textbf{ then } \mathsf{clk}[i] \textbf{ else} \\
& & \quad \textbf{if } \mathsf{S1}_i \textbf{ then } 0 \textbf{ else} \\
& & \quad\quad ((\mathsf{csn}[i] - \mathsf{last})\%\mathsf{C}) \cdot \mathsf{k}_0 + \mathsf{clk}[i] - \mathsf{g} \\
& & \quad \textbf{fi} \\
& & \textbf{fi}
\end{array}
$$

If there has been no synchronization yet ($\mathsf{last} = -1$) then $c(i)$ is just equal to the hardware clock $\mathsf{clk}[i]$. If the synchronizer is in location $\mathsf{S1}_i$, then we know that there has been no tick since the last synchronization, so $c(i)$ is set to 0. Otherwise, $c(i)$ is k_0 times the number of slots since the last synchronization, incremented by the number of ticks in the current slot, minus g to take into account that the hardware clock has been reset to $\mathsf{g}+1$ after the last synchronization.

Now the main invariant result from this section can be stated.

Theorem 3.3.3 *Assume constraints (3.5), (3.6) and (3.7),* $3 \leq \mathsf{N}$ *and assume that some node transmits in the initial slot.*[1] *Then for any network with full connectivity the following invariant assertions hold, for all reachable states and for all* $i, j \in \mathsf{Nodes}$*:*

$$
\begin{array}{rcll}
 & & \mathsf{y} \leq c(i) \cdot \mathsf{max} + \mathsf{x}_i & (3.20) \\
c(i) > 0 & \Rightarrow & \mathsf{y} \geq (c(i) - 1) \cdot \mathsf{min} + \mathsf{x}_i & (3.21) \\
\mathsf{csn}[i] = \mathsf{tsn}[i] \wedge & & & (3.22) \\
(\mathsf{clk}[i] < \mathsf{g} \vee \mathsf{GO\_SEND}_i) & \Rightarrow & \mathsf{last} \neq \mathsf{csn}[i] & (3.23) \\
\mathsf{csn}[i] = \mathsf{tsn}[i] \wedge \mathsf{clk}[i] = \mathsf{g} & \Rightarrow & (\mathsf{GO\_SEND}_i \vee \mathsf{SENDING}_i) & (3.24) \\
\mathsf{csn}[i] = \mathsf{tsn}[i] \wedge \mathsf{clk}[i] > \mathsf{g} & \Rightarrow & \mathsf{last} = \mathsf{csn}[i] & (3.25) \\
\mathsf{SENDING}_i & \Rightarrow & \mathsf{csn}[i] = \mathsf{csn}[j] = \mathsf{last} & (3.26) \\
\mathsf{GO\_SEND}_i & \Rightarrow & \mathsf{csn}[i] = \mathsf{csn}[j] \wedge \mathsf{clk}[i] = \mathsf{g} & (3.27) \\
\mathsf{last} \neq -1 \wedge \mathsf{last} \neq \mathsf{PREV}(\mathsf{csn}[i]) & \Rightarrow & (\mathsf{TX}(\mathsf{csn}[i]) \wedge \mathsf{last} = \mathsf{csn}[i]) & (3.28) \\
\mathsf{TX}(\mathsf{csn}[i]) \wedge \mathsf{clk}[i] = \mathsf{k}_0 - 1 & \Rightarrow & \mathsf{last} = \mathsf{csn}[i] & (3.29) \\
\mathsf{S1}_i & \Rightarrow & \mathsf{clk}[i] < \mathsf{k}_0 - 1 \wedge \mathsf{last} = \mathsf{csn}[i] & (3.30) \\
 & & c(i) \geq 0 & (3.31) \\
\mathsf{last} = -1 & \Rightarrow & \mathsf{csn}[i] = 0 & (3.32)
\end{array}
$$

Proof By induction, using the invariants from Lemma's 3.3.1 and 3.3.2.

For a manual proof see `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/HSV09/`.

[1] The last two assumptions have been made for convenience. They are not needed, even though they are used in the proof.

Invariants (3.20) and (3.21) are the key invariants that relate the values of $c(i)$ and y. Invariant (3.26) implies that the network is synchronized. This is the main correctness property we are interested in. All the other invariants in Theorem 3.3.3 are auxiliary assertions, needed to make the invariant inductive.

3.3.2 On the formal proof

The manual proof of the invariants from the previous subsection has been fully checked using the proof assistant Isabelle [77]. Below some general remarks about the formalization are made and some of the subtleties encountered are discussed. The main motivation for discussing some of the proof details is that this sheds light on the type of reasoning that will be necessary in order to completely automate the verification.

The length of the Isabelle/HOL proof is about 5300 lines, whereas the manual proof is around 1000 lines. Formal proofs are usually longer than their manual counterpart. Wiedijk [101, 21] proposes the *De Bruijn factor* as a way to quantify this difference. This factor basically compares the size of two proof files, compressed using the Unix utility *gzip*. Wiedijk [101] observes that the average De Bruijn factor is about 4. In this case, 4.58 is obtained. This is a bit larger than usual, since the formal proof includes the definition of the UPPAAL model and its semantics, which are not included in the manual proof. All the invariants are also needed to be defined (about 500 lines). In the manual proof, the 12 basic invariants defined in Lemmas 3.3.1 and 3.3.2 are all disposed of by the word "trivial". The formal proof is indeed straightforward but still occupies about 440 lines.

Key aspects of the Isabelle formalization are (1) an alternative definition of function PREV and a proof of lemmas showing particular properties of it, and (2) a formalization of the claim that there are at least three transmitting slots per frame. Common to these two issues is the introduction of the largest slot number in which a message is transmitted. This is the maximum of function tsn and is obtained for node i_{max}. The properties needed are basic facts like $\mathsf{PREV}(s)$ cannot be s or that in the idle period of a frame $\mathsf{PREV}(s)$ equals the transmitting slot of i_{max}, i.e., $\mathsf{tsn}[i_{\mathsf{max}}]$. Altogether, the definition of PREV, the introduction of i_{max}, the formal proof that there are at least three transmitting slots, and the proof of basic properties about these notions occupy about 600 lines.

In the remainder of this section, first i_{max} is formally introduced. Then, the definition of function PREV is rephrased and a sequel of properties of that function is proved. After that, the claim that there are at least three transmitting slots is formalized. Finally, the formal proof is illustrated by two simple but representative examples.

Definition of i_{max} and PREV

As shown in Figure 2.2, a frame is composed of an active period and an idle period. In the active period, there are slots where a node is transmitting and the other

nodes are listening, and also slots where no node is sending and all nodes are listening. Consequently, there is a last slot in which a message is emitted. Let i_{max} be the node that is transmitting in this slot. This transmitting node maximizes function tsn:

$$\mathsf{TX}_{\mathsf{max}}(i_{\mathsf{max}}) \equiv \mathsf{TX}(\mathsf{tsn}[i_{\mathsf{max}}]) \wedge \forall i \neq i_{\mathsf{max}}.\mathsf{tsn}[i_{\mathsf{max}}] > \mathsf{tsn}[i] \tag{3.33}$$

The formal definition of function PREV in Isabelle slightly differs from Equation 3.4. The combination of modulo and the incrementation in the argument does not translate to Isabelle, where functions must be total and proved to terminate. Basically, the modulo is removed and frames are considered to be unbounded. The assumption is still that function tsn returns a natural number strictly less than n. The first basic invariants then prove that parameters take values in their intended domain. Function PREV is the recursive function below:

Definition

$$\begin{array}{lcl} \mathsf{PREV}(0) & = & \mathsf{tsn}[i_{\mathsf{max}}] \\ \mathsf{PREV}(s+1) & = & \textbf{if } \mathsf{TX}(s) \textbf{ then } s \textbf{ else } \mathsf{PREV}(s) \end{array}$$

Properties of PREV

In the formal proof, a sequel of properties showing the structure of a frame is needed. The next lemma asserts that function PREV is constant during the idle period, that is, if slot s is transmitting and all slots from s to y are not transmitting, then $\mathsf{PREV}(y)$ is slot s.

Lemma 3.3.4 $(\mathsf{TX}(s) \wedge y > s \geq 0 \wedge (\forall z.s < z < y \Rightarrow \neg\mathsf{TX}(z))) \Rightarrow \mathsf{PREV}(y) = s$

Proof By induction on s.

From this above lemma it directly follows that after the last transmitting slot, function PREV equals this slot:

Lemma 3.3.5 $\forall y > \mathsf{tsn}[i_{\mathsf{max}}].\mathsf{PREV}(y) = \mathsf{tsn}[i_{\mathsf{max}}]$

Proof By definition i_{max} is such that there is no transmitting slot after it. This fact is used for instantiating Lemma 3.3.4 above.

The aim is to prove that the previous slot of slot s is strictly less than s. Because of the cyclic nature of a frame, this is only true if $s > 0$.

Lemma 3.3.6 $s > 0 \Longrightarrow \mathsf{PREV}(s) < s$

Proof By induction on s.

Another useful lemma asserts that the "PREV" of a transmitting slot cannot be $\mathsf{tsn}[i_{\mathsf{max}}]$.

Lemma 3.3.7 $(s > 0 \wedge \mathsf{TX}(s)) \Longrightarrow \mathsf{PREV}(s) < \mathsf{tsn}[i_{\mathsf{max}}]$

Proof From $\mathsf{TX}(s)$ we obtain j such that $\mathsf{tsn}[j] = s$. By definition of i_{max} we have $\mathsf{tsn}[j] \leq \mathsf{tsn}[i_{\mathsf{max}}]$. Using Lemma 3.3.6, we obtain $\mathsf{PREV}(s) < s$. Hence $\mathsf{PREV}(s) < \mathsf{tsn}[i_{\mathsf{max}}]$.

At least three sending nodes

In the informal case study description [80], it is assumed that for each node there is a transmission slot. Translated to the setting of the model, this means that tsn is a total function from nodes to slots. Interestingly, the Isabelle formalization revealed that the assumption that tsn is total, is never used in the proof. [2] The only assumption that is made is that there are at least three sending nodes.

In the formalization, a predicate $\mathsf{TX}_n(i)$ is introduced which states that for node i there exists a slot s that equals the transmitting slot of node i, that is, node i is a transmitting node. Predicate $\mathsf{TX}_n(i)$ complements predicate $\mathsf{TX}(s)$ defined earlier. Predicate $\mathsf{TX}_n(i)$ is defined as follows:

Definition $\mathsf{TX}_n(i) = \exists s.\mathsf{tsn}[i] = s$

The assumption that there are at least three transmitting slots is formalized by assuming that predicate TX_n holds for nodes 0 to 2.

$$\forall i \leq 2.\mathsf{TX}_n(i) \tag{3.34}$$

Two important facts are derived: $\mathsf{tsn}[i_{\mathsf{max}}]$ is at least 2, and between slot number 0 and slot number $n-1$ there is at least one transmitting slot.

Lemma 3.3.8 $\mathsf{tsn}[i_{\mathsf{max}}] \geq 2 \wedge \exists s.0 < s < n-1 \wedge \mathsf{TX}(s)$

Proof The first part is trivial. Function tsn assigns different slots to different nodes. Together with Equation 3.34 we can derive three distinct nodes – say i, j, k – with distinct transmitting slots ($\mathsf{tsn}[i]$, $\mathsf{tsn}[j]$, $\mathsf{tsn}[k]$). We do a case analysis on their different possible orderings (e.g., $\mathsf{tsn}[i] < \mathsf{tsn}[j] < \mathsf{tsn}[k]$). By definition a slot number is not greater than $n-1$ and positive. Consequently, the "tsn" in the middle of the ordering is strictly positive and strictly less than $n-1$. This shows the second term of the conclusion.

A consequence of Lemma 3.3.8 is that function PREV is at least one for all slots not smaller than $n-1$.

Lemma 3.3.9 $s \geq n-1 \Longrightarrow \mathsf{PREV}(s) \geq 1$

Proof We consider two cases. If $s > n-1$, then $s > \mathsf{tsn}[i_{\mathsf{max}}]$. Moreover there is no transmitting slot between s and $n-1$. So, from Lemma 3.3.4 we obtain $\mathsf{PREV}(s) > \mathsf{tsn}[i_{\mathsf{max}}] > 1$. If $s = n-1$, then we know from Lemma 3.3.8 that there is at least one transmitting node between slot 0 and $n-1$ and $\mathsf{PREV}(s)$ is then at least equal to this slot.

[2] This observation may have practical implications. The results suggest that, at least in certain situations, if nodes have nothing to say they may in fact remain silent.

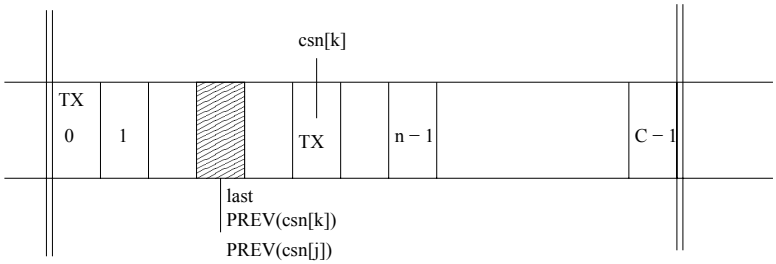

Figure 3.9: Proof example

Proof samples

In the remainder of this section, two examples are presented that show some of the subtleties in the proof. These example illustrates why some of the lemmas introduced earlier are needed, e.g., Lemma 3.3.7 and Lemma 3.3.8.

Example The situation of the first proof sample is pictured in Figure 3.9. This situation appeared in the proof of Invariant 21 and 23 of Theorem 3.3.3. It involves nodes k and an arbitrary different node j. Node k is sending in its current slot number, i.e. we have $\mathsf{csn}[k] = \mathsf{tsn}[k]$ and $\mathsf{TX}(\mathsf{csn}[k])$. The last transmitting slot (depicted in the gray slot) is the previous transmitting slot of both nodes j and k.

$$\mathsf{PREV}(\mathsf{csn}[k]) = \mathsf{PREV}(\mathsf{csn}[j]) = \mathsf{last} \tag{3.35}$$

The goal is to prove that these two nodes agree on the current slot number, i.e., that $\mathsf{csn}[k] = \mathsf{csn}[j]$.

The formal proof needs a case analysis on the relative positions of $\mathsf{csn}[j]$ and $\mathsf{csn}[k]$. Assume $\mathsf{csn}[j] < \mathsf{csn}[k]$. Node k is in a later slot. Because of the cyclic nature of frames, we must consider two cases: $\mathsf{csn}[j] = 0$ and $\mathsf{csn}[j] > 0$.

If $\mathsf{csn}[j] = 0$, by definition of PREV we have $\mathsf{PREV}(\mathsf{csn}[j]) = \mathsf{tsn}[i_{\mathsf{max}}]$, and $\mathsf{PREV}(\mathsf{csn}[k]) = \mathsf{tsn}[i_{\mathsf{max}}]$ also. From Lemma 3.3.7 we have $\mathsf{PREV}(\mathsf{csn}[k]) < \mathsf{csn}[k]$. So, we have $\mathsf{tsn}[i_{\mathsf{max}}] < \mathsf{csn}[k] = \mathsf{tsn}[k]$. By definition of i_{max}, this is clearly impossible.

The case when $\mathsf{csn}[k] < \mathsf{csn}[j]$ is similar. We know that $\mathsf{TX}(\mathsf{csn}[k])$, so k is transmitting and has to be in the active region, i.e., $\mathsf{csn}[k] < n$. We do not have such information for $\mathsf{csn}[j]$ and need to consider extra cases: $\mathsf{csn}[j] < n$ and $\mathsf{csn}[j] \geq n$.

Example Now the proof of Invariant 3.22 in Theorem 3.3.3 is illustrated. This invariant assumes that node i is in a transmitting slot. We have $\mathsf{csn}[i] = \mathsf{tsn}[i]$, hence $\mathsf{TX}(\mathsf{csn}[i])$. It also considers that node i is either in state WAIT with $\mathsf{clk}[i] < \mathsf{g}$) or in state GO_SEND. (By basic Invariant 3.15 node i cannot be in state SENDING.) In brief, node i is about to send a message. The conclusion asserts that the last slot with a synchronization is not the current slot ($\mathsf{last} \neq \mathsf{csn}[i]$). Before

the first synchronization, last is negative ($\mathsf{last} = -1$) and the conclusion holds as any csn is a nonnegative number (basic Invariant 3.10). Before a start_message action, node i is in state GO_SEND with its clock equal to g. Variable $\mathsf{clk}[i]$ is not modified by this action. Hence the invariant is trivially true in the target state because its premises are false. The case of a tick action is more complicated.

We only consider the end of the current slot ($\mathsf{csn}[i]$). The situation is as follows. Node i is in state WAIT with its clock counting the last tick of a slot ($\mathsf{clk}[i] = \mathsf{k_0} - 1$). After the tick action the clock is reset to 0, a new slot starts ($\mathsf{csn}'[i] = (\mathsf{csn}[i] + 1)\%C$), and other variables are left unchanged, in particular $\mathsf{last}' = \mathsf{last}$. The last transmitting slot is the previous transmitting slot of i ($\mathsf{last} = \mathsf{PREV}(\mathsf{csn}[i])$). The case where $\mathsf{csn}[i] = 0$ or $\mathsf{csn}[i] = n - 1$ is illustrated.

The latter implies that $\mathsf{csn}'[i] = 0$. The conclusion rewrites to $\mathsf{PREV}(n-1) \neq 0$. This directly follows from the fact that PREV is at least one (Lemma 3.3.9).

If $\mathsf{csn}[i] = 0$, we have $\mathsf{csn}'[i] = 1$. The conclusion rewrites to $\mathsf{PREV}(0) \neq 1$. By definition, $\mathsf{PREV}(0) = \mathsf{tsn}[i_{\mathsf{max}}]$ and the conclusion follows from the first term of Lemma 3.3.8 ($\mathsf{tsn}[i_{\mathsf{max}}] \geq 2$).

3.4 Uppaal Analysis Results: Networks with 4 Nodes

In the two previous sections, the correctness of the clock synchronization protocol for networks with full connectivity was studied. In practice, however, wireless sensor networks are rarely fully connected. A fully parametric analysis of the protocol for arbitrary network topologies will be quite involved.

In order to illustrate some of the complications, a small script is written to explore all possible network topologies with 4 nodes. As explained in Section 3.1, it is easy to model arbitrary network topologies in Uppaal by appropriate instantiation of the neighbor function. For parameters $\mathsf{k_0} = 15$, $\mathsf{C} = 6$ and $\mathsf{n} = 4$, Uppaal was used to find out for each topology for which guard time and min/ max ratio the synchronization property was satisfied. This took 30 hours for the $4^6 = 4096$ possible topologies. The results of the model checking experiments for the connected networks of Figure 3.10 in which communication is symmetric, is summarized in Table 3.6. As in Section 3.2, min and max in this table are the smallest consecutive natural numbers for which the model with the values assigned to C, n, $\mathsf{k_0}$ and g is synchronized. As expected, topology number 6, requires the highest guard time of all networks with 4 nodes. We observe that the more connected the network is, the lower the guard time can be.

When communication is not symmetric, that is, it may occur that some node A receives messages from a node B but not vice versa, the clock synchronization behavior becomes highly unpredicatable and depends on time slot numbers assigned to each node. Table 3.7 summarizes the analysis results for the networks depicted in Figure 3.11. Surprisingly, network 1 allows for smaller guard times

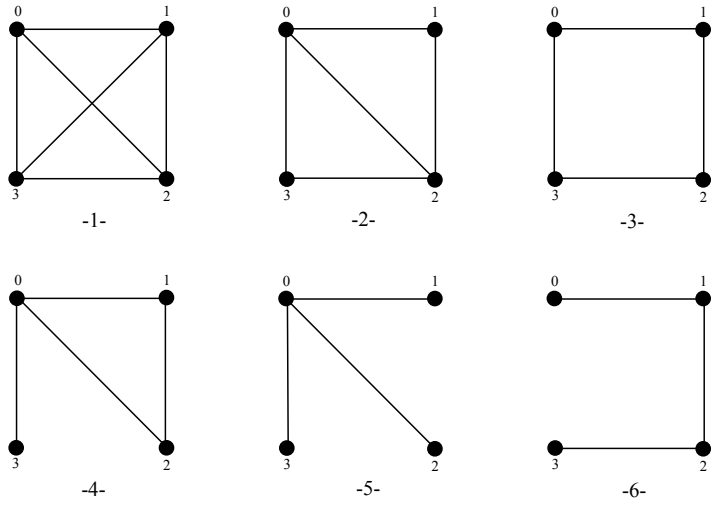

Figure 3.10: Connected networks with 4 nodes

ID	Diameter	Time Slot Numbers	Guard Time	min/max Ratio
1	1	any	2	44/45
2	2	{0,1,2,3}	3	58/59
		{0,2,1,3}	3	73/74
3	2	{0,1,2,3}	3	58/59
		{0,2,1,3}	3	73/74
4	2	{0,1,2,3}	3	58/59
		{0,1,3,2}	3	43/44
		{0,3,1,2}	3	73/74
5	2	any	3	88/89
6	3	{0,1,2,3}	4	88/89
		{0,3,1,2}	3	43/44

Table 3.6: Analysis results for the networks of Figure 3.10

than network 3, even though it has fewer links.

3.5 Uppaal Analysis Results: Line Topologies

Since the experiments indicate, among the symmetric topologies, line topologies have the worst clock synchronization behavior, Uppaal was used for model checking of some further instances of the protocol that involve line topologies, that is, connected networks in which each node is connected to exactly two other nodes, except for two nodes that only have a single neighbor.

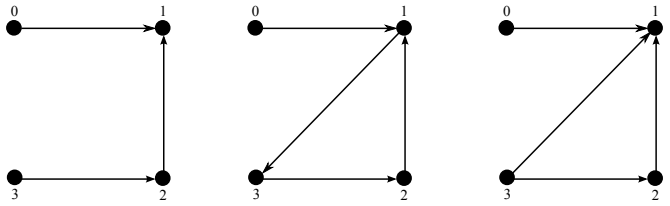

Figure 3.11: Some directed network topologies with 4 nodes

Time Slot Numbers	ID	Guard Time	min/max Ratio
{0,1,2,3}	1	2	74/75
	2	4	83/84
	3	3	38/39
{1,0,2,3}	1	2	45/46
	2	4	87/88
	3	3	43/44
{2,1,3,0}	1	2	30/31
	2	3	87/88
	3	3	73/74
{1,2,0,3}	1	2	74/75
	2	not synchronized at all	
	3	2	74/75

Table 3.7: Analysis results for the networks of Figure 3.11

A 3-node network with line topology was defined in Uppaal and the behavior of the system for different variable valuations was checked. It turns out that, unlike the fully connected network with three nodes (see Table 3.2), the network will not always remain synchronized for $\mathsf{g} = 2$, even when the clocks are perfect. Table 3.8 lists some of the verification results. On the left the results are given for a line network of size 3 and on the right those for a clique network of size 3. If we compare these results then we see that, in order to keep the network synchronized, the hardware clocks in a line topology must be more accurate than the hardware clocks in a fully connected network (i.e., the min/max ratio must be closer to 1 if we want the network to be synchronized). Intuitively, the reason is that in a line topology the frequency of synchronization for each node is less than that in a fully connected network.

In order to maintain synchronization, a line topology requires more accurate hardware clocks and a larger guard time. The claim is that, for a fixed value of the guard time, the network may become unsynchronized if we keep increasing the number of nodes. In fact, the claim is that for a line topology of size N, the guard

C	6	8	10	12
n	4	4	4	4
k_0	10	10	10	10
g	3	3	3	3
min	58	78	98	118
max	59	79	99	119

C	6	8	10	12
n	4	4	4	4
k_0	10	10	10	10
g	3	3	3	3
min	19	29	39	49
max	20	30	40	50

Table 3.8: Results for line network of size 3 (left) and for clique network of size 3 (right)

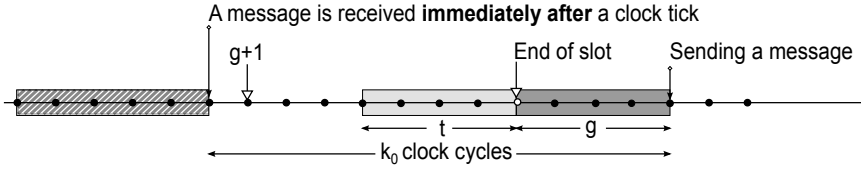

Figure 3.12: Maximum distance between two consecutive clock synchronization events

time g should be at least N.

Model checking of synchronization for line topology entails exploring a state space that grows exponentially with the number of nodes. In order to reduce the state space, only networks with perfect clocks are considered. However, even with perfect clocks, UPPAAL can only handle networks with at most 8 nodes. Table 3.9 shows the resource usage of UPPAAL required for model checking of networks with line topologies. A Sun Fire X4440 machine with 4 Opteron 8356 2.3 Ghz quad-core processors and 128 Gb DDR2-667 memory was used. One processor on this machine needs about half an hour to establish that a line network with 8 nodes is synchronized if the guard time is 8.

The reason why we run into state space explosions even in a setting with perfect clocks, is that race conditions are possible involving arrival of messages and ticking of hardware clocks. As a result even a network with perfect clocks will not necessarily remain synchronized for any parameter valuation. Figures 3.12 and 3.13 illustrate how race conditions may affect the time interval between two synchronization events in the model. (For simplicity, we assume $g = t$.) We consider the case where a node is the receiver in one slot and the sender in the next slot. We know that the sender sends a message when the value of its clock equals g, and that the receiver resets its clock counter to $g + 1$ at the first clock tick after receiving the message. Figure 3.12 shows that a synchronization signal is received immediately *after* a clock tick at the receiver. In this scenario, the receiver waits a full clock cycle before resetting its clock counter to $g + 1$. Figure 3.13 illustrates a

Nodes	g	Collision	Time	Memory
2	1	yes	0.008 s	240852 KB
	2	no	0.039 s	240852 KB
3	2	yes	0.160 s	240852 KB
	3	no	0.200 s	240852 KB
4	3	yes	1.007 s	240852 KB
	4	no	1.012 s	240852 KB
5	4	yes	2.570 s	240852 KB
	5	no	2.587 s	240852 KB
6	5	yes	17.000 s	240852 KB
	6	no	18.006 s	240852 KB
7	6	yes	163.154 s	326892 KB
	7	no	173.922 s	336672 KB
8	7	yes	1624.481 s	2328572 KB
	8	no	1681.874 s	2451884 KB

Table 3.9: CPU time and memory usage of Uppaal for line networks of different sizes

different scenario in which a synchronization signal is received immediately *before* the receiver clock ticks and the receiver immediately resets its clock counter to $\mathsf{g}+1$. We see that the length of the time interval between two synchronization events in the first scenario is one clock cycle longer than that in the second scenario.

Now it will be shown that in a line network of size N and with guard time $\mathsf{g} = \mathsf{N}-1$, there is a reachable state in which the network is no longer synchronized. Thus synchronization of line networks can only be ensured if $\mathsf{g} \geq \mathsf{N}$. In the examples, $\mathsf{tsn}[i] = i\%3$, that is, the transmission slot number of node i equals i modulo 3. Note that, for line topologies, this allocation of transmission slot numbers satisfies the conditions (3.2) and (3.3) defined at the end of Section 3.1. Figures 3.14, 3.16 and 3.17 illustrate three abstract error scenarios, extracted from concrete counterexamples produced by Uppaal, resulting in a loss of synchronization. Figure 3.14 applies to the case in which N modulo 3 equals 0, Figure 3.16 to the case in which N modulo 3 equals 1, and Figure 3.17 to the case in which N modulo 3 equals 2. The example of Figure 3.14 is explained in detail. The other two scenarios are similar.

The scenario of Figure 3.14 consists of two "staircases". One "fast" staircase has steps with minimum time between synchronizations (using the mechanism of Figure 3.13), where a synchronization signal is received immediately before the receiver clock ticks and the receiver resets its clock counter to $\mathsf{g}+1$ immediately,

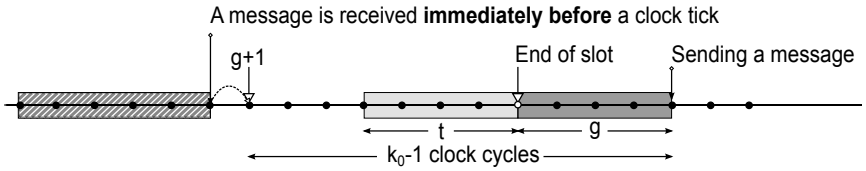

Figure 3.13: Minimum distance between two consecutive clock synchronization events

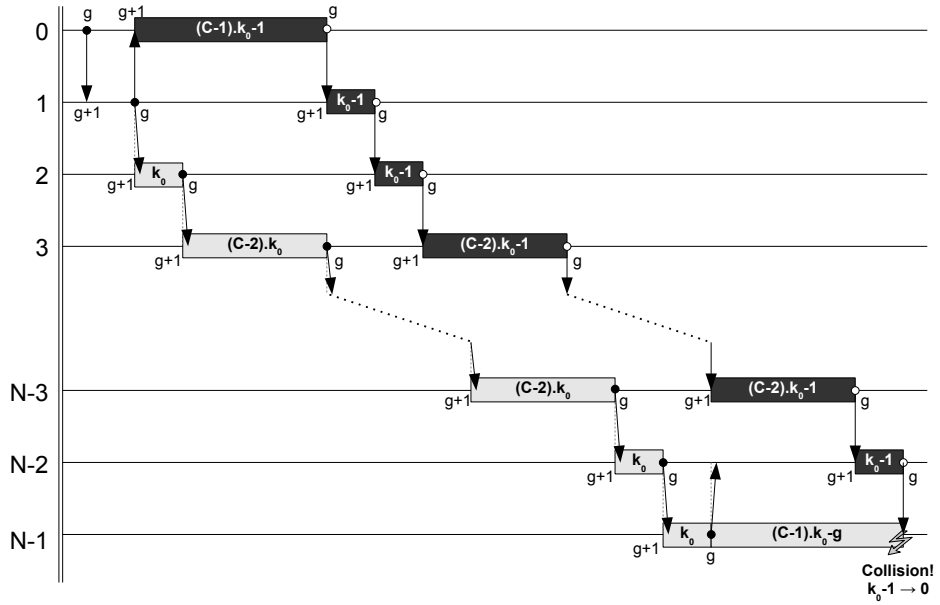

Figure 3.14: Error scenario for line topologies when $N \mod 3 = 0$

while the other "slow" staircase has steps with the maximum time between synchronizations (using the mechanism of Figure 3.12), where a synchronization signal is received immediately after the receiver clock ticks, and it takes an additional clock tick before resetting the clock is reset to $g+1$. Both staircases start from the same point, viz. when node number 1, the second node in the line, sends messages to its neighboring nodes 0 and 2. After $N-1$ steps the two staircases join again when node $N-2$ tries to communicate with node $N-1$. At that point, node $N-2$ has gone through g time units since its previous synchronization and is about to send a message to node $N-1$. However, node $N-1$ is about to make a clock tick and enter its new time slot, which is convenient for receiving the message from its neighbor. Synchronization is lost when node $N-2$ starts sending before node $N-1$ ticks.

This proof, in sum, shows that for each network of line topology in which guard

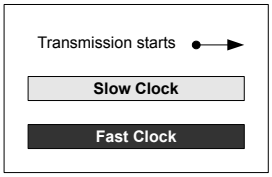

Figure 3.15: Guide to figures 3.14, 3.16 and 3.17

time is less than N clock cycles, based on the given error scenarios, the network may fail to keep synchronization. Accordingly, to guarantee synchronization in a network of line topology with N nodes, guard time should be equal to or grater than N.

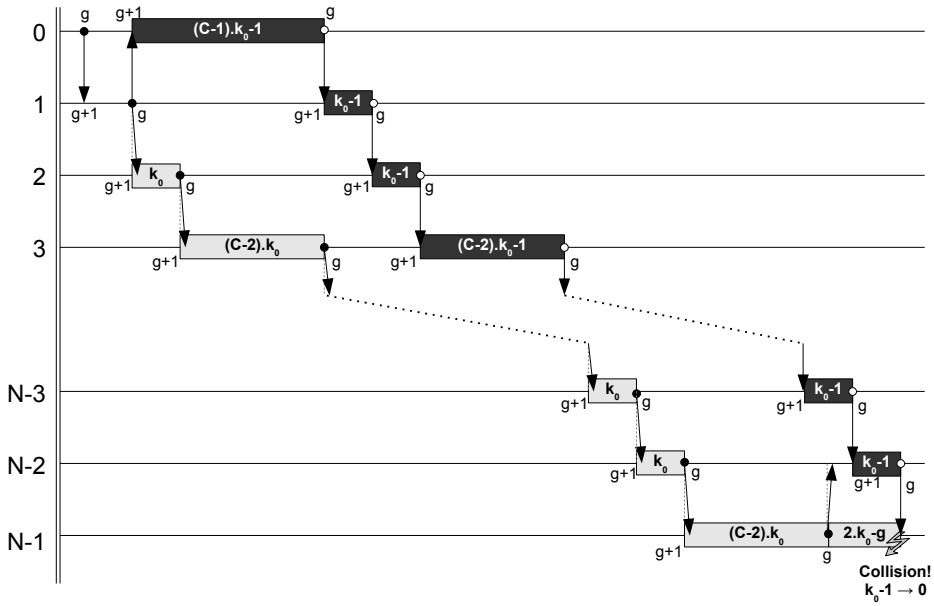

Figure 3.16: Error scenario when N $\mod 3 = 1$

3.6 Conclusion

This chapter demonstrated the application of timed automata model checking and Uppaal in modeling and analysis of a synchronization protocol for wireless sensor networks, provided by the Chess eT International B.V.

We have seen timed automata model checking led to discovery of some interesting error scenarios for line topologies: for any instantiation of the parameters, the protocol will eventually fail if the network grows.

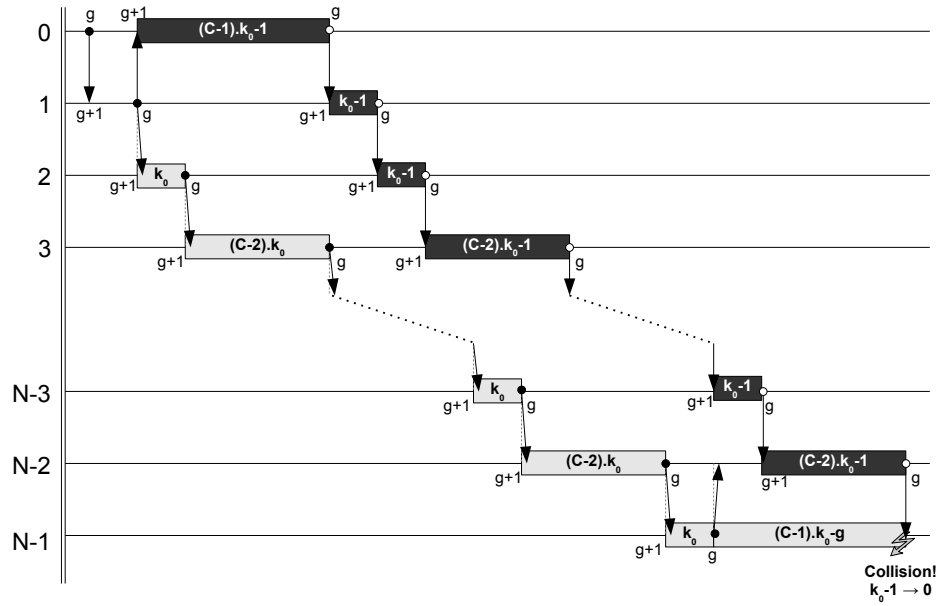

Figure 3.17: Error scenario when N mod 3 = 2

Moreover, in this chapter, a parametric verification for the very restrictive case of cliques (network with full connectivity) was presented. Indeed, model checking was used to find the key error scenarios that underly the parameter constraints for correctness. The parameter constraints were then proved to be sufficient and necessary for the network to be synchronized. Afterwards, the correctness of the manual invariant proof was checked by automatic theorem proving.

Despite its limitations, UPPAAL proved to be indispensable for the formal analysis of the Chess protocol. Modeling the protocol in terms of timed automata is natural, the graphical user interface helped to visualize the models, the simulator was of great help during the initial validation of the model, and the ability of UPPAAL to generate counterexamples and to replay them in the simulator was helpful for finding the parameter constraints that are needed for correctness. Since UPPAAL does not support parametric verification, the sufficiency of the constraints was proved manually. But UPPAAL was also helpful in checking the validity of various invariants for instances of the model, and obtaining confidence in their correctness before embarking on the (long and tedious) invariant proofs.

Using UPPAAL, we have only been able to analyze models of some really small networks, and in order to carry out the analysis making some drastic simplifying assumptions were inevitable. Clearly, there is a lot of room for improving timed automata model checking technology, enabling the analysis of larger and dynamic network topologies. Nevertheless, it is concluded that the ability of model checkers to find worst-case error scenarios appears to be quite useful in this application

domain.

In practical applications of WSNs, cliques rarely occur and therefore the verification results should primarily be seen as a first step towards a correctness proof for arbitrary and dynamically changing network topologies. Nevertheless, these results give us an upper bound on allowable clock drift of a generic WSN.

Methodologically, the approach of this research is similar to the study of the Biphase Mark Protocol [97], which also uses UPPAAL to analyze instances of the protocol and a theorem prover for the full parametric analysis. Theorem provers have been frequently and successfully applied for the analysis of clock synchronization protocols, see for instance [86, 88]. An interesting research challenge is to synthesize (or prove the correctness of) the parameter constraints for the Chess protocol fully automatically. Recently, some approaches have been presented by which, for instance, the (parametric) Biphase Mark Protocol can be verified fully automatically [20, 96]. Very interesting also is the work of [22, 39] on parameterized verification (using the SMT based tool MCMT) of networks with an arbitrary number of identical timed automata. However, it seems that these approaches are not powerful enough (yet) to handle this WSN protocol in which the number N of sensor nodes is not fixed, and the parameter constraints and the length of the corresponding counterexamples depend on N. Moreover, in the case of this WSN algorithm the parameter constraints involve a product of three parameters, whereas the mentioned techniques can only handle linear constraints on the parameters. Finally, these new tools still lack the graphical user interface and expressive input language of UPPAAL, which are key features that enable the application of formal methods in practice.

Chapter 4

Modeling of Median Algorithm for Synchronization of Wireless Sensor Networks

In this chapter, a detailed model of the synchronization algorithm used in the implementation of the Chess wireless sensor network is presented. The algorithm, named Median, is an extension of an algorithm proposed by Tjoa et al [91], and the idea is that in every frame each node computes its phase error to any of its direct neighbors. After the last active slot, each node adjusts its clock by the median of the phase errors of its immediate neighbors. A detailed model of the Chess Median algorithm is presented in this chapter using the input language of Uppaal. The objective is to construct a model that comes as close as possible to the specification of the clock synchronization algorithm presented in [80]. This chapter is organized as follows. In Section 4.1, the Median algorithm is presented in detail. Section 4.2 describes the Uppaal model of Median algorithm. In Section 4.3, the analysis results are described. Finally, in Section 4.4, some conclusions are drawn.

The Uppaal model described in this chapter is available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/chess09/`.

4.1 The Median Protocol

In this section additional details are provided about the Median protocol as it has been implemented by Chess.

4.1.1 The Synchronization Algorithm

In each frame, each node broadcasts one message to its neighbors. The timing of this message is used for synchronization purposes: a receiver may estimate the clock value of a sender based on the time when the message is received. Thus there is no need to send around (logical) clock values. In the current implementation of Chess, clock synchronization is performed once per frame using the following algorithm [7, 100]:

1. In its sending slot, a node broadcasts a packet which contains its transmission slot number.

2. Whenever a node receives a message it computes the `phase error`, that is the difference (number of clock cycles) between the expected receiving time and the actual receiving time of the incoming message. Note that the difference between the sender's sending slot number (which is also the current slot number of the sender) and the current slot number of the receiving node must also be taken into account when calculating the phase errors.

3. After the last active slot of each frame, a node calculates the `offset` from the phase errors of all incoming messages in this frame with the following algorithm:

   ```
   if (number of received messages == 0)
      offset = 0;
   else if (number of received messages <= 2)
      offset = gain *
                  the phase error of the first received message;
   else
      offset = gain * the median of all phase errors
   ```

   Here `gain` is a coefficient with value 0.5, used to prevent oscillation of the clock adjustment.

4. During the sleeping period, the frame length of each node is adjusted by the computed `offset` obtained from step 3.

In situations when two networks join, it is possible that the phases of these networks differ so much that the nodes in one network are in active slots whereas the nodes in the other network are in sleeping slots and vice versa. In this case, no messages can be exchanged between two networks. Therefore in the Chess design, a node will send an extra message in one (randomly selected) sleeping slot to increase the chance that networks can communicate and synchronize with each other. This slot is called the synchronization slot and the message is in the same format as in the transmission slot. The extreme value of `offset` can be obtained when two networks join: it may occur that the `offset` is larger than half the total

number of clock cycles of sleeping slots in a frame. Chess uses another algorithm called `join` to handle this extreme case. For simplicity, the `join` algorithm is not modeled, so joining of networks and synchronization messages are not covered in this research.

4.1.2 Guard Time

The desirable correctness condition for gMAC is that whenever a node is sending all its neighbors are in receiving mode. However, at the moment when a node enters its TX slot it cannot be guaranteed, due to the phase errors, that its neighbors have entered the corresponding RX slot. This problem is illustrated in Figure 4.1 (a). Given two nodes 1 and 2, if a message is transmitted during the entire sending slot of node 1 then this message may not be successfully received by node 2 because of the imperfect slot alignment. Taking the clock of node 1 as a reference, the clock of node 2 may drift backwards or forwards. In this situation, node 1 and node 2 may have a different view of the current slot number within the time interval where node 1 is sending a message.

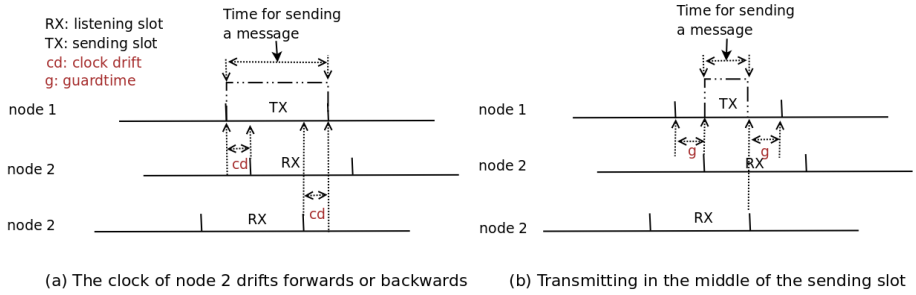

Figure 4.1: The need for introducing guard times

To cope with this problem, messages are not transmitted during the entire sending slot but only in the middle, as illustrated in Figure 4.1 (b). Both at the beginning and at the end of its sending slot, node 1 does not transmit for a preset period of g clock ticks, in order to accomodate the forwards and backwards clock drift of node 2. Therefore, the time available for transmission equals the total length of the slot minus 2g clock ticks.

4.1.3 Radio Switching Time

The radio of a wireless sensor node can either be in sending mode, or in receiving mode, or in idle mode. Switching from one mode to another takes time. In the current implementation of the Chess gMAC protocol, the radio switching time is around 130μsec. The time between clock ticks is around 30μsec and the guard time g is 9 clock ticks. Hence, in the current implementation the radio switching time

is smaller than the guard time, but this may change in future implementations. If the first slot in a frame is an RX slot, then the radio is switched to receiving mode some time before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, in order to keep the implementation simple, the radio is switched to the receiving mode only at the start of the RX slot. Therefore messages arriving in such receiving slots may not be fully received. This issue may also affect the performance of the synchronization algorithm.

4.2 Uppaal Model

In this section, the Uppaal model of the gMAC protocol is described.

A finite, fixed set of wireless nodes $\mathsf{Nodes} = \{0, \ldots, \mathsf{N}-1\}$ is assumed. The behavior of an individual node $\mathsf{id} \in \mathsf{Nodes}$ is described by five timed automata **Clock**(id), Receiver(id), Sender(id), **Synchronizer**(id) and Controller(id). Figure 4.2 shows how these automata are interrelated. All components interact with the clock, although this is not shown in Figure 4.2. Automaton **Clock**(id) models the hardware clock of node id, automaton Sender(id) the sending of messages by the radio, automaton Receiver(id) the receiving part of the radio, automaton **Synchronizer**(id) the synchronization of the hardware clock, and automaton Controller(id) the control of the radio and the clock synchronization.

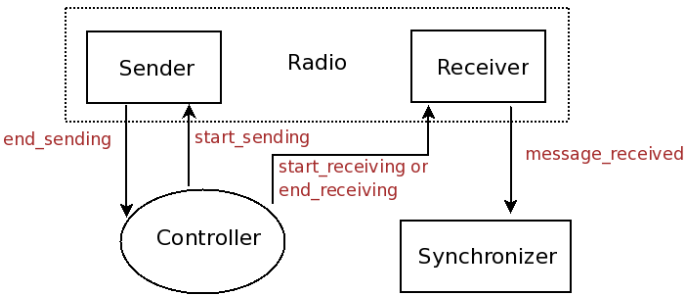

Figure 4.2: Message flow in the model

Table 4.1 lists the parameters that are used in the model (constants in Uppaal terminology), together with some basic constraints. The domain of all parameters is the set of natural numbers. Now, the five automaton templates used in the model are described.

Clock Timed automaton **Clock**(id) models the behavior of the hardware clock of node id. The automaton is shown in Figure 4.3. At the start of the system state variable csn[id], that records the current slot number, is initialized to $\mathsf{C}-1$, that is, to the last sleeping slot. Hardware clocks are not perfect and so a

Parameter	Description	Constraints
N	number of nodes	$1 < N$
C	number of slots in a time frame	$0 < C$
n	number of active slots in a time frame	$0 < n \leq C$
tsn[id]	TX slot number for node id	$0 \leq tsn[id] < n$
k_0	number of clock ticks in a time slot	$0 < k_0$
g	guard time	$0 < g$
r	radio switch time	$0 \leq r$
min[id]	minimal time between two clock ticks of node id	$0 < min[id]$
max[id]	maximal time between two clock ticks of node id	$min[id] \leq max[id]$

Table 4.1: Protocol parameters

minimal time min[id] and a maximal time max[id] between successive clock ticks are assumed. Integer variable clk[id] records the current value of the hardware clock. For convenience (and to reduce the size of the state space), the hardware clock is assumed to be reset at the end of each slot, that is after k_0 clock ticks. Also, a state variable csn[id], which records the current slot number of node id, is updated each time at the start of a new slot.

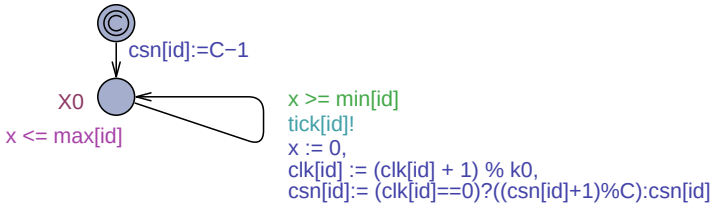

Figure 4.3: Automaton **Clock**[id]

Sender The sending behavior of the radio is described by the automaton Sender[id] shown in Figure 4.4. The behavior is rather simple. When the controller asks the sender to transmit a message (via a start_sending[id] signal), the radio first switches to sending mode (this takes r clock ticks) and then transmits the message (this takes $k_0 - 2 \cdot g$ ticks). Immediately after the message transmission has been completed, an end_sending[id] signal is sent to the controller to indicate that the message has been sent.

Receiver The automaton Receiver[id] models the receiving behavior of the radio. The automaton is shown in Figure 4.5. Again the behavior is rather simple. When the controller asks the receiver to start receiving, the receiver first switches to receiving mode (this takes r ticks). After that, the receiver may receive messages

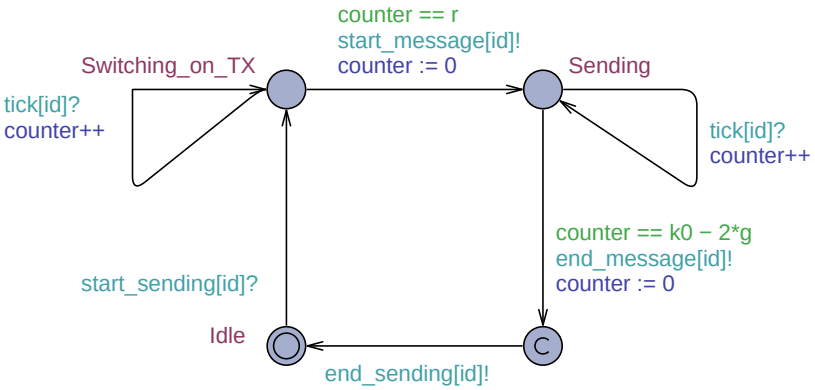

Figure 4.4: Automaton Sender[id]

from all its neighbors. A function neighbor is used to encode the topology of the network: neighbor(j, id) holds if messages sent by j can be received by id. Whenever the receiver detects the end of a message transmission by one of its neighbors, it immediately informs the synchronizer via a message_received[id] signal. At any moment, the controller can switch off the receiver via an end_receiving[id] signal.

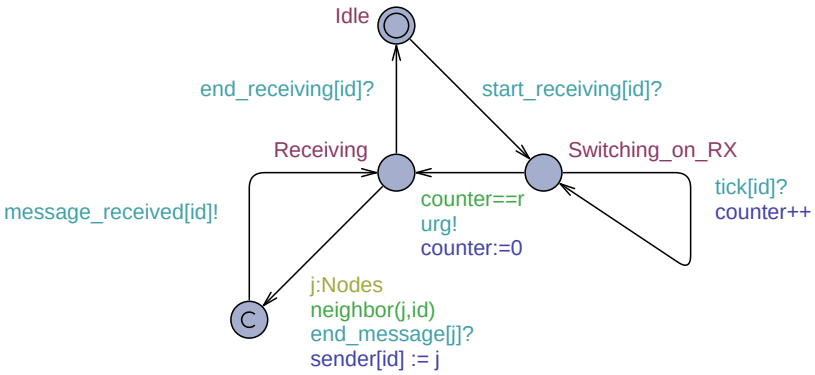

Figure 4.5: Automaton Receiver[id]

Controller The task of the Controller[id] automaton, displayed in Figure 4.6, is to put the radio in sending and receiving mode at the appropriate moments. Figure 4.7 shows the definition of the predicates used in this automaton. The radio should be put in sending mode r ticks before message transmission starts (at time g in the transmission slot of id). If $r > g$ then the sender needs to be activated $r - g$ ticks before the end of the slot that precedes the transmission slot. Otherwise, the sender must be activated at tick $g - r$ of the transmission slot. If the first slot in

a frame is an RX slot, then the radio is switched to receiving mode r time units before the start of the frame to ensure that the radio will receive during the full first slot. However if there is an RX slot after the TX slot then, as described in Section 4.1.3, the radio is switched to the receiving mode only at the start of the RX slot. The controller stops the radio receiver whenever either the last active slot has passed or the sender needs to be switched on.

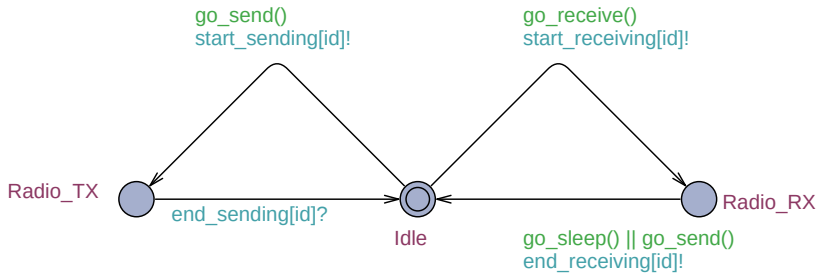

Figure 4.6: Automaton Controller[id]

```
bool go_send(){return (r>g)
 ?((csn[id]+1)%C==tsn[id] && clk[id]==k0-(r-g))
 :(csn[id]==tsn[id] && clk[id]==g-r);}

bool go_receive(){return
(r>0 && 0!=tsn[id] && csn[id]==C-1 && clk[id]==k0-r)
|| (r==0 && 0!=tsn[id] && csn[id]==0)
|| (0<csn[id] && csn[id]<n && csn[id]-1==tsn[id]);}

bool go_sleep(){return csn[id]==n;}
```

Figure 4.7: Predicates used in Controller[id]

All the channels used in the Controller[id] automaton (start_sending, end_sending, start_receiving, end_receiving and synchronize) are urgent, which means that these signals are sent at the moment when the transitions are enabled.

Synchronizer Finally, automaton **Synchronizer**[id] is shown in Figure 4.8. The automaton maintains a list of phase differences of all messages received in the current frame, using a local array phase_errors. Local variable msg_counter records the number of messages received. Whenever the receiver gets a message from a neighboring node (message_received[id]), the synchronizer computes and stores the phase difference using the function store_phase_error at the next clock tick. Here the phase difference is defined as the expected time at which the message transmission ends (`tsn[sender] * k0 + k0 - g`) minus the actual time at which the message transmission ends (`csn[id] * k0 + clk[id]`), counting from

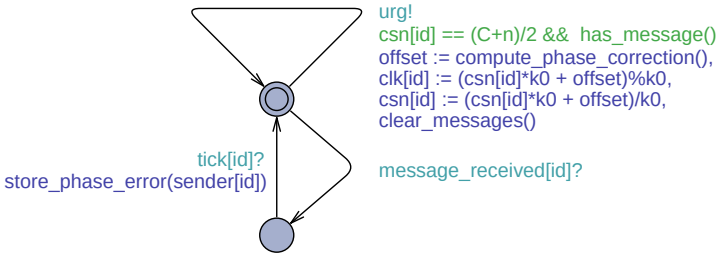

Figure 4.8: Automaton **Synchronizer**[id]

```
void store_phase_error(int sender)
{
phase_errors[msg_counter] =
   (tsn[sender] * k0 + k0 - g)
        - (csn[id] * k0 + clk[id]);
msg_counter++
}
```

Figure 4.9: Function used in **Synchronizer**[id]

the start of the frame. The complete definition is listed in Figure 4.9. Recall that in the model transmission delays are ignored.

As explained in Section 4.1.1, the synchronizer computes the value of the phase correction (offset) and adjusts the clock during the sleeping period of a frame.[1] Hence, in order to decide in which slot the synchronization should be performed, the maximal phase difference between two nodes should be known. In this model, it is assumed no joining of networks occurs. When a node receives a message from another node, the phase difference computed using this message will not exceed the length of an active period. Otherwise one of these two nodes will be in sleeping period while the other is sending, hence no message can be received at all. In practice, the number of sleeping slots is much larger than the number of active slots. Therefore it is safe to perform the adjustment in the middle of sleeping period because the desired property described above holds. When the value of `gain` is smaller than 1 the maximal phase difference will be even smaller.

The function of compute_phase_correction implements exactly the algorithm listed in Section 4.1.1.

[1] Actually, in the implementation the offset is used to compute the corrected *wakeup time*, that is the moment when the next frame will start [100]. In this model the clock is reset, but this should be equivalent.

4.3 Analysis Results

In this section, the verification results for simple instances of the model described in Section 4.2 are presented. The following invariant properties were checked using the UPPAAL model checker:

```
INV1 : A[] forall (i: Nodes) forall (j : Nodes)
   SENDER(i).Sending && neighbor(i,j) imply RECEIVER(j).Receiving

INV2 : A[] forall (i:Nodes) forall (j:Nodes) forall (k:Nodes)
   SENDER(i).Sending && neighbor(i,k) &&
   SENDER(j).Sending && neighbor(j,k) imply i == j

INV3 : A[] not deadlock
```

The first property states that always when some node is sending, all its neighbors are listening. The second property states that never two different neighbors of a given node are sending simultaneously. The third property states that the model contains no deadlock, in the sense that in each reachable state at least one component can make progress. The three invariants are basic sanity properties of the gMAC protocol, at least in a setting with a static topology and no transmission failures.

UPPAAL was used on a Sun Fire X4440 machine (with 4 Opteron 8356 2.3 Ghz quad-core processors and 128 Gb DDR2-667 memory) to verify instances of the model with different number of nodes, different network topologies and different parameter values. Table 4.2 lists some of the verification results, including the resources UPPAAL needed to verify if the network is synchronized or not. In all experiments, $\mathsf{C} = 10$ and $\mathsf{k}_0 = 29$.

Clearly, the values of network parameters, in particular clock parameters min and max, affect the result of the verification. Table 4.2 shows several instances where the protocol is correct for perfect clocks ($\mathsf{min} = \mathsf{max}$) but fails when the ratio $\frac{\mathsf{min}}{\mathsf{max}}$ is decreased. It is easy to see that the protocol will always fail when $\mathsf{r} \geq \mathsf{g}$. Consider any node i that is not the last one to transmit within a frame. Right after its sending slot, node i needs r ticks to get its radio into receiving mode. This means that — even with perfect clocks — after g ticks another node already has started sending even though the radio of node i is not yet receiving. Even when $\mathsf{r} < \mathsf{g}$, the radio switching time has a clear impact on correctness: the larger the radio switching time is, the larger the guard time has to be in order to ensure correctness. Using UPPAAL, it is possible to fully analyze line topologies with at most seven nodes if all clocks are perfect. For larger networks UPPAAL runs out of memory. A full parametric analysis of this protocol will be challenging, also due to the impact of the network topology and the selected slot allocation. Using UPPAAL, it is discovered that for certain topologies and slot allocations the Median algorithm may always violate the above correctness assertions, irrespective of the

N/n	Topology	g	r	$\frac{\text{min}}{\text{max}}$	CPU Time	Peak Memory Usage	Sync
3/3	clique	2	0	1	1.944 s	24,180 KB	YES
3/3	clique	2	0	$\frac{100,000}{100,001}$	492.533 s	158,064 KB	NO
3/3	line	2	0	1	1.068 s	68,144 KB	YES
3/3	line	2	0	$\frac{100,000}{100,000}$	441.308 s	68,144 KB	NO
3/3	clique	3	0	1	1.851 s	28,040 KB	YES
3/3	clique	3	0	$\frac{100,000}{100,001}$	575.085 s	272,312 KB	NO
3/3	line	3	0	1	1.05 s	23,348 KB	YES
3/3	line	3	0	$\frac{451}{452}$	29.545 s	148,012 KB	NO
3/3	line	3	0	$\frac{452}{453}$	35.257 s	148,012 KB	YES
3/3	clique	3	2	1	1.827 s	24,184 KB	YES
3/3	clique	3	2	$\frac{100,000}{100,001}$	109.633 s	26,056 KB	NO
3/3	line	3	2	1	1.052 s	22,916 KB	YES
3/3	line	3	2	$\frac{100,000}{100,001}$	82.383 s	78,360 KB	NO
3/3	clique	4	2	$\frac{100,000}{100,001}$	533.345 s	350,504 KB	NO
3/3	line	4	2	$\frac{100,000}{100,001}$	414.201 s	53,752 KB	NO
4/4	clique	3	0	1	231.297 s	1,437,643 KB	YES
4/4	clique	3	0	$\frac{450}{451}$	Memory Exhausted		
4/3	line	3	0	1	4.749s s	94,748 KB	YES
4/3	line	3	0	$\frac{450}{451}$	Memory Exhausted		
4/4	clique	3	2	1	229.469 s	1,438,368 KB	YES
4/4	clique	3	2	$\frac{100,000}{100,001}$	14,604.531 s	2,317,040 KB	NO
4/3	line	3	2	1	4.738 s	94,748 KB	YES
4/3	line	3	2	$\frac{100,000}{100,001}$	1,923.655 s	1,264,844 KB	YES
5/5	clique	3	0	1	Memory Exhausted		
5/3	line	3	0	1	46.54 s	249,976 KB	YES
6/3	line	3	0	1	508.19 s	2,316,416 KB	YES
7/3	line	3	0	1	Memory Exhausted		

Table 4.2: Model checking experiments

choice of the guard time. For example, in a 4 node-network with clique topology and min and max of 100.000 and 100.001, respectively, if the median of the clock drifts of a node becomes -1, the median algorithm divides it by 2 and generates 0 for clock correction value and indeed no synchronization happens. If this scenario repeats in three consecutive time frames for the same node, that node runs $\mathsf{g} = 3$ clock cycles behind and gets out of sync.

Another example in which the algorithm may fail is displayed in Figure 4.10. This network has 4 nodes, connected by a line topology, that send in slots 1, 2, 3, 1, respectively. Since all nodes have at most two neighbors, the Median

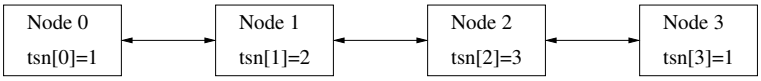

Figure 4.10: A problematic network configuration

algorithm prescribes that nodes will correct their clocks based on the first phase error that they see in each frame. For the specific topology and slot allocation of Figure 4.10, this means that node 0 adjusts its clock based on phase errors of messages it gets from node 1, node 1 adjusts its clock based on messages from node 0, node 2 adjusts its clock based on messages from node 3, and node 3 adjusts its clock based on messages from node 2. Hence, for the purpose of clock synchronization, we have two disconnected networks! Thus, if the clock rates of nodes 0 and 1 are lower than the clock rates of nodes 2 and 3 by just an arbitrary small margin, then two subnetworks will eventually get out of sync. These observations are consistent with results obtained using Uppaal. If, for instance, we set $\mathsf{min}[\mathsf{id}] = 99$ and $\mathsf{max}[\mathsf{id}] = 100$, for all nodes id then neither `INV1` nor `INV2` holds. In practice, it is unlikely that the above scenario will occur due to the fact that in the implementation slot allocation is random and dynamic. Due to regular changes of the slot allocation, with high probability node 1 and node 2 will now and then adjusts their clocks based on messages they receive from each other.

However, variations of the above scenario may occur in practice, even in a setting with dynamic slot allocation. In fact, the above synchronization problem is also not restricted to line topologies. A subset C of nodes in a network is called a *community* if each node in C has more neighbors within C than outside C [75]. For *any* network in which two disjoint communities can be identified, the Median algorithm allows for scenarios in which these two parts become unsynchronized. Due to the median voting mechanism, the phase errors of nodes outside a community will not affect the nodes within this community, independent of the slot allocation. Therefore, if nodes in one community A run slow and nodes in another community B run fast then the network will become unsynchronized eventually, even in a setting with infinitesimal clock drifts. Figure 4.11 gives an example of a network with two communities.

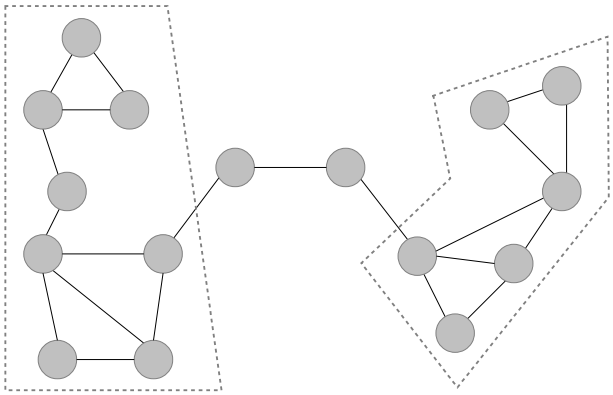

Figure 4.11: Another problematic network topology with two communities

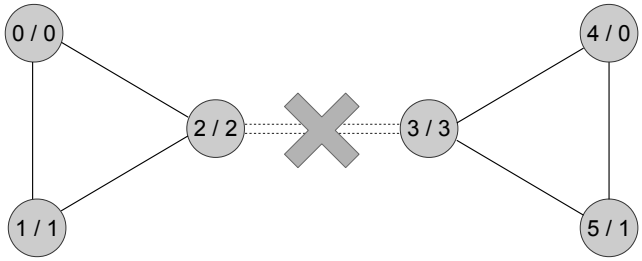

Figure 4.12: A network with two communities that was analyzed using Uppaal

Using Uppaal, instances of the simple network with two communities displayed in Figure 4.12 were analyzed. The numbers on the vertices are the node identifiers and the transmission slot numbers, respectively. Table 4.3 summarizes the results of the model checking experiments.

It is still needed to explore how realistic these counterexamples are. Network topologies with multiple communities occur in many WSN applications. Nevertheless, in practice the gMAC protocol appears to perform quite well for static networks. It might be that problems do not occur so often in practice due to the probabilistic distributions of clock drift and jitter.

4.4 Conclusion

This chapter represented a detailed Uppaal model of relevant parts of the clock synchronization algorithm implemented in a wireless sensor network developed by Chess [80, 100].

Using Uppaal, it was established that in certain cases a static, fully synchro-

g	r	Fast Clock Cycle Length	Slow Clock Cycle Length	CPU Time	Peak Memory Usage
2	0	1	1	Memory Exhausted	
2	0	99	100	457.917 s	2,404,956 KB
2	1	99	100	445.148 s	2,418,032 KB
3	0	99	100	416.796 s	2,302,548 KB
3	2	1	1	Memory Exhausted	
3	2	99	100	22.105 s	83,476 KB
3	2	451	452	798.121 s	3,859,104 KB
3	2	452	453	Memory Exhausted	
4	0	99	100	424.935 s	2,323,004 KB
4	1	99	100	464.503 s	2,462,176 KB
4	2	99	100	420.742 s	2,323,952 KB

Table 4.3: Model checking experiments of a network with two communities

nized network may eventually become unsynchronized if the Median algorithm is used, even in a setting with infinitesimal clock drifts.

In another research, presented in chapter 3, another synchronization algorithm for WSN was presented that does not have the correctness problems of the Median algorithm. However, that algorithm has never been tested in practice. Advantages of that approach are (a) unlike the Median approach and its variants we need almost no guard time at the end of a sending slot (2 clock ticks suffice instead of 9 ticks in the current implementation), and (b) the computational overhead becomes essentially zero.

Assegei [7] proposed and simulated three alternative algorithms, to be used instead of the Median algorithm, in order to achieve decentralized, stable and energy-efficient synchronization of the Chess gMAC protocol. It should be easy to construct Uppaal models for Assegei's algorithms: basically, only the definition of the `compute_phase_correction` function should be modified.

Overall, starting from the Uppaal model introduced in this chapter it should be relatively easy to construct models for alternative synchronization algorithms in order to explore their properties.

Chapter 5

Conclusion of Part One

Wireless sensor networks are beginning to be deployed at an accelerated pace due to their wide range of application potential in areas such as target detection and tracking, environmental monitoring, industrial process monitoring, and tactical systems. Low power capacities of sensor nodes makes the design of medium access protocols pretty challenging. One of the greatest challenges in the design of the MAC layer is to find suitable mechanisms for clock synchronization: we must ensure that whenever some node is sending all its neighbors are listening.

Many clock synchronization protocols have been proposed for WSNs, see e.g. [90, 35, 91, 70, 7, 63, 79]. However, these protocols (with the exception of [91, 7] and possibly [79]) involve a computation and/or communication overhead that is unacceptable given the extremely limited resources (energy, memory, clock cycles) available within the Chess nodes. In most of these protocols, clocks are synchronized to an accurate real-time standard like Coordinated Universal Time (UTC). In [90] an overview of this type of protocols is presented. However, these protocols are based on the exchange of time stamp messages, and for the Chess WSN this creates an unacceptable computation and communication overhead. It is possible to come up with more efficient algorithms, since for the MAC layer a weak form of clock synchronization suffices: a node only needs to be synchronized to its immediate neighbors, not to faraway nodes or to UTC. Fan & Lynch [35] study the *gradient clock synchronization (GCS)* problem, in which the difference between any two network nodes' clocks must be bounded from above by a non-decreasing function. Thus nearby nodes must be closely synchronized but faraway nodes are allowed to be more loosely synchronized. In the approach of Fan & Lynch [35], nodes compute logical clock values based on their hardware clocks and message exchanges, and the goal is to synchronize the nodes' logical clocks as closely as possible, while satisfying certain validity conditions. Logical clocks have been introduced by Lamport [56] to totally order the events in a distributed system. A key property of Lamport's logical clocks is that they never run backwards: their value can only increase. In fact, Fan & Lynch [35] assume a constant lower bound

on clock speed. Also Meier & Thiele [70] and Pussente & Barbosa [79], who adapt the work of Fan & Lynch to the setting of wireless sensor networks, make a similar assumption (with minimal clock rates $\frac{1}{2}$ and $\frac{1}{\mathsf{D}}$, respectively, where D is the network diameter). For certain applications of WSNs it is important to have Lamport style logical clocks. For example, if two sensor nodes observe a moving object, then logical clocks allow one to establish the object's direction by determining which node observed the object first [70]. However, for the MAC layer there is no need to compute a total order on events: we only need to ensure that whenever one node is sending all neighbors are listening. Since it is allowed to set back clocks, the lower bounds of [35, 70] do not apply in this case.

Quasimodo research on the Chess WSN case study had a significant impact on the design of the network [82]. But equally important, it also provided major challenges for research on theory and algorithms for model checking. Most industrial applications of Uppaal thus far involve small networks with a fixed topology. The size and dynamic nature of the Chess WSN, and the resulting complexity of clock synchronization, provides a major challenge for model checking technology that goes beyond what we have seen in other Uppaal case studies. Another challenge raised by the present research is in the area of parametric model checking: the parameter constraints that were derived in chapter 3 for clique topologies are nonlinear, and appear to be beyond reach of existing algorithms for parametric model checking. We have seen, even the analysis of a basic clock synchronization algorithm for an industrial WSN platform turns out to be quite difficult.

Meier & Thiele [70] provide a lower bound for the achievable synchronization quality in sensor networks, but no algorithms that attain or come close to this bound. Pussente & Barbosa [79] also proposed a clock synchronization algorithm that achieves an $O(1)$ worst-case skew between the logical clocks of neighbors, but this cannot be applied in the TDMA based setting of the Chess algorithm. Basic assumptions of [70, 79] are that (a) messages sent between neighbors are always delivered instantaneously, and (b) consecutive communications between any two neighbors in the same direction are no farther apart in time than some given time d. Pussente & Barbosa [79] derive a strict upper bound of $c + 2(1 + 2\hat{\rho})d$ on the difference between the clocks of neighboring nodes, where $c > 0$ is a constant and $\hat{\rho} \in [0, 1)$ is the maximal clock drift. But since this bound exceeds $2d$ and in a TDMA setting d basically equals the length of a frame, the algorithm of [79] is unable to guarantee that whenever some node is sending all its neighbors are listening.

All in all, a fundamental open question is to establish an impossibility result along the lines of Fan & Lynch [35] for the setting of the Chess MAC layer in which clocks can be set back. The MyriaNed algorithm analyzed in chapter 3 appears to perform well for networks with a small diameter, but the results of section 3.5 show that performance may degrade if the diameter increases. However, the counterexample scenarios described in this thesis require a very specific combination of events and clock drifts; hence it may not appear so often in practice. Chess has

already demonstrated twice that a network of the size in the order of 1000 nodes works without encountering these error scenarios.

Wireless sensor networks also constitute a potentially very important but also extremely challenging application area for formal methods. For quantitative formal methods, one challenge is to come up with the right abstractions that will facilitate verifying larger instances of the model.

Using state-of-the-art model checking technology, we have only been able to analyze models of some really small networks. In order to carry out the analysis making some drastic simplifying assumptions were inevitable. Clearly, there is a lot of room for improving timed automata model checking technology, enabling the analysis of larger and dynamic network topologies. Nevertheless, it is concluded that the ability of model checkers to find worst-case error scenarios appears to be quite useful in this application domain. In particular, error scenarios— found using UPPAAL by exploring simple models of small networks — are reproducible in real implementations of larger networks [81].

The use of simulations is essential for providing insight into the robustness and usefulness of MAC layer protocols, also because occasional flaws of the MAC layer protocols may be resolved by the redundancy of the gossip layer. However, this research asserts that it is unlikely that simulation techniques will be able to produce worst case counterexamples, such as the example of Figure 3.14 that was produced by UPPAAL. Work of [24] also shows that one has to be extremely careful in using the results of MANET simulators.

Wireless sensor networks algorithms pose many challenges for probabilistic model checkers and specification tools such as PRISM [54] and CaVi [36]. A first challenge is to make a more detailed, probabilistic model of radio communication that involves the possibility of message loss. Another challenge is to consider dynamic slot allocation. In the research of this thesis, a fixed slot allocation was assumed. However, in the actual implementation of Chess, a sophisticated probabilistic algorithm is used for dynamic slot allocation. Formal analysis of this algorithm would be very interesting. Another simplifying assumption made in this research is that the network topology is fixed. A probabilistic model in which nodes may join or leave will be more realistic. Finally, the gossiping algorithms used by the Chess network are intrinsically probabilistic in nature. Formal analysis of the gossip layer is a largely unexplored research field [11]. Practical obstacles for the application of probabilistic model checkers to the Chess case study are the limited expressivity of the input language of existing tool, and the small network sizes that can be handled. An interesting alternative approach for some of the problems in this area is the use of mean-field analysis, as proposed by Bakhshi et al [12].

Several other researches report on the application of UPPAAL for the analysis of protocols for wireless sensor networks, see e.g. [37, 36, 95, 43]. In [103], UPPAAL is also used to automatically test the power consumption of wireless sensor networks. This research confirms the conclusions of [37, 95]: despite the small

number of nodes that can be analyzed, model checking provides valuable insight in the behavior of protocols for wireless sensor networks, insight that is complementary to what can be learned through the application of simulation and testing or theorem proving.

Part II

Automata Learning through Counterexample-guided Abstraction Refinement

Chapter 6

Introduction to Part Two

Model-based system development is becoming an increasingly important driving force in the software and hardware industry. In this approach, models become the primary artifacts throughout the engineering lifecycle of computer-based systems. Requirements, behavior, functionality, construction and testing strategies of computer-based systems are all described in terms of models. Models are not only used to reason about a system, but also used to allow all stakeholders to participate in the development process and to communicate with each other, to generate implementations, and to facilitate reuse. The construction of models typically requires significant manual effort, implying that in practice often models are not available, or become outdated as the system evolves. Tools that are able to infer state machine models automatically by systematically "pushing buttons" and recording outputs have numerous applications in different domains. For instance, they support understanding and analyzing legacy software, regression testing of software components [48], protocol conformance testing based on reference implementations, reverse engineering of proprietary/classified protocols, fuzz testing of protocol implementations [29], and inference of botnet protocols [27]. Automated support for constructing behavioral models of implemented components would therefore be extremely useful.

The problem to build a state machine (automata) model of a system by providing inputs to it and observing the outputs resulting, which often referred to as black-box system identification, is fundamental and has been studied for decades. A major challenge is to let computers perform this task in a rigorous manner for systems with large numbers of states. Moore [74] first proposed the problem of learning automata, provided an exponential algorithm, and proved that this problem is inherently exponential. Many techniques for constructing models from observation of component behavior have been proposed, for instance in [6, 84, 83, 45].

The most efficient such techniques use the setup of *active learning*, where a model of a system is learned by actively performing experiments on that system.

In other words, within the setting of active learning, a learner interacts with a teacher. This problem was addressed by Angluin [6] in her L^* algorithm for learning of finite state automata (FSA). Niese [76] adapted the L^* algorithm for active learning of deterministic Mealy machines. This algorithm has been further optimized in [83]. The assumption, in the algorithm, is that the teacher knows a deterministic Mealy machine $\mathcal{M}$, and the learner, initially, knows the action signature (the sets of input and output symbols I and O). The learner's task is to learn a Mealy machine that is equivalent to $\mathcal{M}$. The teacher reacts to two types of queries: –output queries ("what is the output generated in response to input $i \in I^*$?") and equivalence queries ("is a hypothesized machine $\mathcal{H}$ correct, i.e., equivalent to the machine $\mathcal{M}$?"). The learner always records the current state q of Mealy machine $\mathcal{M}$. In response to output query i, the current state is updated to q' and answer o is returned to the learner. At any point the learner can "reset" the teacher, that is, change the current state back to the initial state of $\mathcal{M}$. The answer to an equivalence query $\mathcal{H}$ is either yes (in case $\mathcal{M} \approx \mathcal{H}$) or no (in case $\mathcal{M} \not\approx \mathcal{H}$). Furthermore, with every negative equivalence query response, the teacher will provide the learner with a counterexample, that is an input sequence $u \in I^*$ such that $obs_{\mathcal{M}}(u) \neq obs_{\mathcal{H}}(u)$. This algorithm has been implemented in the LearnLib tool [83, 46, 71], the winner of the 2010 Zulu competition on regular inference. In practice, when a real implementation is used instead of an idealized teacher, the implementation cannot answer equivalence queries. Therefore, LearnLib "approximates" such queries by generating a long test sequence that is computed by standard methods such as state cover, transition cover, W-method, and the UIO method among which fully randomized testing turn out to be most effective. (see [58]).

LearnLib has been applied successfully to learn various kinds of systems, such as computer telephony integrated (CTI) systems [48]. Nevertheless, a lot of further research will be required to make automata based learning tools suitable for routine use on industrial case studies. An issue is the extension of automata learning techniques to nondeterministic systems (see e.g. [102]). Furthermore, in practice, the characteristic of Mealy machines that each input corresponds to exactly one output is overly restrictive. Sometimes several inputs are required before a single output occurs, sometimes a single input triggers multiple outputs, etc. In [3], Aarts & Vaandrager addressed this problem by developing a method for active learning of I/O automata. The I/O automata of Lynch & Tuttle [67, 68] and Jonsson [52] constitute a popular modeling framework which does not suffer from the restriction that inputs and outputs have to alternate. Aarts & Vaandrager used LearnLib, and their idea was to place a transducer in between the IOA teacher and the Mealy machine learner, which translates concepts from the world of I/O automata to the world of Mealy machines, and vice versa. The transducer and Mealy machine learner together then implement an IOA learner. Another important issue, clearly, is the development of abstraction techniques in order to be able to learn much larger state spaces (see [1], also for further references). State-

of-the-art methods for learning automata such as LearnLib are currently able to only learn automata with at most in the order of 10.000 states. Hence, powerful abstraction techniques are needed to apply these methods to practical systems. There are many reasons to expect that by combining ideas from verification, model-based testing and automata learning, it will become possible to learn models of realistic software components with state-spaces that are many orders of magnitude larger than what state-of-the-art tools can currently handle. See e.g. [18, 64, 83].

The second part of this dissertation presents a framework for automata learning. In chapter 7 this framework is described in full mathematical details. Thereafter, in chapter 8, the tool Tomte is introduced. Tomte enables learning of a restricted class of parametrized Mealy machines (i.e. Mealy machines, for which each action contains a few parameters), which are called scalarset Mealy machines. in scalarset Mealy machines, one can test for equality of data parameters, but no operations on data are allowed. In the rest of this introduction, a brief history of this research is presented together with a rough description of the idea proposed in this thesis. Furthermore, Tomte functionality is shortly explained.

6.1 History Dependent Abstraction

Applying existing automata learning methods on realistic applications is typically achievable only through abstraction. Dawn Song at al [27], for instance, succeeded to infer models of realistic botnet command and control protocols by placing an emulator between botnet servers and the learning software, which concretizes the alphabet symbols into valid network messages and sends them to botnet servers. When responses are received, the emulator does the opposite — it abstracts the response messages into the output alphabet and passes them on to the learning software. The idea of an intermediate component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning.

Within process algebra [19], the most prominent abstraction operator is the τ_I operator from ACP, which renames actions from a set I into the internal action τ. In order to establish that an implementation *Imp* satisfies a specification *Spec*, one typically proves $\tau_I(Imp) \approx Spec$, where $\approx$ is some behavioral equivalence or preorder that treats τ as invisible. In state based models of concurrency, such as TLA+ [57], the corresponding abstraction operator is existential quantification, which hides certain state variables. Both τ_I and $\exists$ abstract in a way that does not depend on the history of the computation. In practice, however, we frequently describe and reason about reactive systems in terms of history dependent abstractions. For instance, most of us have dealt with the following protocol: "If you forgot your password, enter your email and user name in the form below. You will then receive a new, temporary password. Use this temporary password to login and immediately select a new password." Here, essentially, the huge name spaces for user names and passwords are abstracted into small sets with abstract values

such as “temporary password” and “new password”. The choice which concrete password is mapped to which abstract value depends on the history, and may change whenever the user selects a new password.

History dependent abstractions are the key for scaling methods for active learning of automata to realistic applications. History dependent abstractions can be described formally using the state operator known from process algebra [9], but this operator has been mostly used to model state bearing processes, rather than as an abstraction device. Implicitly, history dependent abstractions play an important role in the work of Pistore et al [73, 38]: whereas the standard automata-like models for name-passing process calculi are infinite-state and infinite-branching, they provide models using the notion of a history dependent automaton which, for a wide class of processes (e.g. finitary π-calculus agents), are finite-state and may be explored using model checking techniques. Aarts, Jonsson and Uijen [1] formalized the concept of history dependent abstractions within the context of automata learning. Inspired by ideas from predicate abstraction [66] and abstract

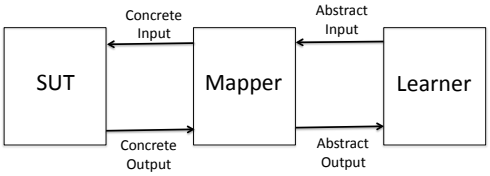

Figure 6.1: Active learning with an abstraction mapping.

interpretation [30], they defined the notion of a *mapper* $\mathcal{A}$, which is placed in between the teacher or system-under-test (SUT), described by a Mealy machine $\mathcal{M}$, and the learner. The mapper transforms the concrete actions of $\mathcal{M}$ (in a history dependent manner) into a small set of abstract actions. Each mapper $\mathcal{A}$ induces an abstraction operator $\alpha_{\mathcal{A}}$ that transforms a Mealy machine over the concrete signature into a Mealy machine over the abstract signature. A teacher for $\mathcal{M}$ and a mapper for $\mathcal{A}$ together behave line a teacher for $\alpha_{\mathcal{A}}(\mathcal{M})$. Hence, by interacting with the mapper component, the learner may learn an abstract Mealy machine $\mathcal{H}$ that is equivalent ($\approx$) to $\alpha_{\mathcal{A}}(\mathcal{M})$. Mapper $\mathcal{A}$ also induces a concretization operator $\gamma_{\mathcal{A}}$. The main technical result of [1] is that, under certain assumptions, $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$ implies $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$.

Aarts et al [1] demonstrated the feasibility of their approach by learning models of fragments of realistic protocols such as SIP and TCP [1], and the new biometric passport [2]. The learned SIP model, for instance, is an extended finite state machine with 29 states, 3741 transitions, and 17 state variables with various types (booleans, enumerated types, (long) integers, character strings,..). This corresponds to a state machine with an astronomical number of states and transitions, which is thus far, fully out of reach of current automata learning techniques.

Despite its success, the theory of [1] has several limitations when facing real-world case studies. Here is an example: In a (deterministic) Mealy machine, each sequence of input actions uniquely determines a corresponding sequence of output actions. This means that the login protocol that was described above cannot be modeled in terms of a Mealy machine, since a single input (a request for a temporary password) may lead to many possible outputs (one for each possible password). The theory of chapter 7 applies to interface automata that are determinate in the sense of Milner [72]. In a determinate interface automaton multiple output actions may be enabled in a single state, which makes it straightforward to model the login protocol. In order to learn the resulting model, it is crucial to define an abstraction that merges all outputs that are enabled in a given state to a single abstract output.

In brief, the theory presented in chapter 7, improved the framework of [1] in four important directions: (a) interface automata instead of the more restricted Mealy machines, (b) the concept of a learning purpose, which allows one to restrict the learning process to relevant behaviors only, (c) a richer class of abstractions, which includes abstractions that over-approximate the behavior of the system-under-test, and (d) a conceptually superior approach for testing correctness of the hypotheses that are generated by the learner.

6.2 Tomte

Chapter 8 presents an algorithm to construct fully automatically mappers for a restricted class of extended finite state machines, called scalarset Mealy machines, in which one can test for equality of data parameters, but no operations on data are allowed. The notion of a scalarset data type originates from model checking, where it is been used by Ip & Dill for symmetry reduction [49]. The algorithm of chapter 8 is implemented in a tool named Tomte, after the creature that shrank Nils Holgersson into a gnome and (after numerous adventures) changed him back to his normal size again.

For the whole research, LearnLib is used as the basic learning tool and therefore the abstraction of the SUT may not exhibit any nondeterminism: if it does then LearnLib crashes. On the other hand, when abstraction is applied, nondeterminism arises naturally: it may occur that the behavior of a SUT is fully deterministic but that due to the mapper (which, for instance, abstracts from the precise value of certain input parameters), the system appears to behave nondeterministically from the perspective of the learner. In case of nondeterminism, abstraction has to be refined in order to dissolve it. This procedure is formalized and the construction of the mapper is described in terms of a counterexample guided abstraction refinement (CEGAR) procedure, similar to the approach developed by Clarke et al [28] in the context of model checking. This is exactly what has been done repeatedly during the manual construction of the abstraction mappings in the case studies of [1].

Using the prototype tool implementation Tomte, models of several realistic software components, including the biometric passport and the SIP protocol were learned *fully automatically*. The Tomte tool and all the models used in the experiments are available via `http://www.italia.cs.ru.nl/`.

Chapter 7

A Theory of History Dependent Abstractions for Learning Interface Automata

This chapter offers a general theory of history dependent abstractions for learning interface automata. In section 7.1, the primary prerequisites are presented. Section 7.2 gives a detailed account of automata learning. Next, mappers and their functionality are introduced in section 7.3. Thereafter, the theory of history dependent abstractions for learning interface automata is fully described in section 7.4. Finally, section 7.5 concludes the work.

7.1 Preliminaries

7.1.1 Interface automata

In this chapter reactive systems are modeled by a simplified notion of *interface automata* [32], essentially labeled transition systems with input and output actions.

Definition An *interface automaton (IA)* is a tuple $\mathcal{I} = \langle I, O, Q, q^0, \rightarrow \rangle$ where

- I and O are disjoint sets of input and output actions, respectively,
- Q is a non-empty set of states,
- $q^0 \in Q$ is the initial state, and
- $\rightarrow \subseteq Q \times (I \cup O) \times Q$ is the transition relation.

We write $q \xrightarrow{a} q'$ if $(q, a, q') \in \rightarrow$. An action a is *enabled* in state q, denoted $q \xrightarrow{a}$, if $q \xrightarrow{a} q'$ for some state q'. We extend the transition relation to sequences by defining, for $\sigma \in (I \cup O)^*$, $\rightarrow_*$ to be the least relation that satisfies, for $q, q', q'' \in Q$ and $a \in I \cup O$,

- $q \xrightarrow{\epsilon}_* q$, and
- if $q \xrightarrow{\sigma}_* q'$ and $q' \xrightarrow{a} q''$ then $q \xrightarrow{\sigma a}_* q''$.

Here ϵ is used to denote the empty sequence. A state q is said to be *reachable* if $q^0 \xrightarrow{\sigma}_* q$, for some σ. We write $q \xrightarrow{\sigma}_*$ if $q \xrightarrow{\sigma}_* q'$, for some state q'. $\sigma \in (I \cup O)^*$ is said to be a *trace* of $\mathcal{I}$ if $q^0 \xrightarrow{\sigma}_*$, and write $Traces(\mathcal{I})$ for the set of traces of $\mathcal{I}$.

A *bisimulation* on $\mathcal{I}$ is a symmetric relation $R \subseteq Q \times Q$ such that $(q^0, q^0) \in R$ and

$$(q_1, q_2) \in R \wedge q_1 \xrightarrow{a} q_1' \quad \Rightarrow \quad \exists q_2' : q_2 \xrightarrow{a} q_2' \wedge (q_1', q_2') \in R.$$

Two states $q, q' \in Q$ are said to be *bisimilar*, denoted $q \sim q'$, if there exists a bisimulation on $\mathcal{I}$ that contains (q, q'). Recall that relation $\sim$ is the largest bisimulation and that $\sim$ is an equivalence relation [72].

Interface automaton $\mathcal{I}$ is said to be:

- *finite* if Q, I, and O are finite sets.
- *finitary* if I and O are finite and there exists a bisimulation $R \subseteq Q \times Q$ that is an equivalence relation with finitely many equivalence classes.
- *deterministic* if for each state $q \in Q$ and for each action $a \in I \cup O$, whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' = q''$.
- *determinate* [72] if for each reachable state $q \in Q$ and for each action $a \in I \cup O$, whenever $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ then $q' \sim q''$.
- *output-determined* if for each reachable state $q \in Q$ and for all output actions $o, o' \in O$, whenever $q \xrightarrow{o}$ and $q \xrightarrow{o'}$ then $o = o'$.
- *behavior-deterministic* if $\mathcal{I}$ is both determinate and output-determined.
- *active* if each reachable state enables an output action.
- *output-enabled* if each state enables each output action.
- *input-enabled* if each state enables each input action.

An *I/O automaton (IOA)* is an input-enabled IA. Our notion of an I/O automaton is a simplified version of the notion of IOA of Lynch & Tuttle [68] in which the set of internal actions is empty, the set of initial states has only one member, and the equivalence relation is trivial.

7.1.2 The ioco relation

A state q of $\mathcal{I}$ is *quiescent* if it enables no output actions. Let δ be a special action symbol. In this article, we only consider IAs $\mathcal{I}$ in which δ is not an input action. The *δ-extension* of $\mathcal{I}$, denoted $\mathcal{I}^\delta$, is the IA obtained by adding δ to the set of output actions, and δ-loops to all the quiescent states of $\mathcal{I}$. Write $O^\delta = O \cup \{\delta\}$. The following lemma easily follows from the definitions.

Lemma 7.1.1 *Let $\mathcal{I}$ be an IA with outputs O. Then*

1. *$\mathcal{I}^\delta$ is active,*
2. *$\mathcal{I}^\delta$ is an IOA iff $\mathcal{I}$ is an IOA,*
3. *if $\mathcal{I}$ is determinate then $\mathcal{I}^\delta$ is determinate,*
4. *if $\delta \notin O$ and $\mathcal{I}^\delta$ is determinate then $\mathcal{I}$ is determinate,*
5. *$\mathcal{I}^\delta$ is output-determined iff $\mathcal{I}$ is output-determined, and*
6. *if $\mathcal{I}$ is behavior-deterministic then $\mathcal{I}^\delta$ is behavior-deterministic.*

Write $out_{\mathcal{I}}(q)$, or just $out(q)$ if $\mathcal{I}$ is clear from the context, for $\{a \in O \mid q \xrightarrow{a}\}$, the set of output actions enabled in state q. For $S \subseteq Q$ a set of states, write $out_{\mathcal{I}}(S)$ for $\bigcup\{out_{\mathcal{I}}(q) \mid q \in S\}$. Write $\mathcal{I}$ **after** σ for the set $\{q \in Q \mid q^0 \xrightarrow{\sigma}_* q\}$ of states of $\mathcal{I}$ that can be reached via trace σ.

The next technical lemma easily follows by induction on the length of trace σ.

Lemma 7.1.2 *Suppose $\mathcal{I}$ is a determinate IA. Then, for each $\sigma \in Traces(\mathcal{I}^\delta)$, all states in $\mathcal{I}^\delta$* **after** *σ are pairwise bisimilar.*

Let $\mathcal{I}_1 = \langle I_1, O_1, Q_1, q_1^0, \rightarrow_1\rangle$, $\mathcal{I}_2 = \langle I_2, O_2, Q_2, q_2^0, \rightarrow_2\rangle$ be IAs with $I_1 = I_2$ and $O_1^\delta = O_2^\delta$. Then $\mathcal{I}_1$ and $\mathcal{I}_2$ are *input-output conforming*, denoted $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$, if

$$\forall \sigma \in Traces(\mathcal{I}_2^\delta) : out(\mathcal{I}_1^\delta \textbf{ after } \sigma) \subseteq out(\mathcal{I}_2^\delta \textbf{ after } \sigma).$$

Informally, an implementation $\mathcal{I}_1$ is **ioco**-conforming to specification $\mathcal{I}_2$ if any experiment derived from $\mathcal{I}_2$ and executed on $\mathcal{I}_1$ leads to an output from $\mathcal{I}_1$ that is allowed by $\mathcal{I}_2$. The **ioco** relation is one of the main notions of conformance in model-based black-box testing [92, 93].

7.1.3 XY-simulations

In the technical development of this chapter, a major role is played by the notion of an XY-simulation. Below we recall the definition of XY-simulation, as introduced in [3], and establish three (new) technical lemma's.

Let $\mathcal{I}_1 = \langle I, O, Q_1, q_1^0, \rightarrow_1\rangle$ and $\mathcal{I}_2 = \langle I, O, Q_2, q_2^0, \rightarrow_2\rangle$ be IAs with the same sets of input and output actions. Write $A = I \cup O$ and let $X, Y \subseteq A$. An *XY-simulation* from $\mathcal{I}_1$ to $\mathcal{I}_2$ is a binary relation $R \subseteq Q_1 \times Q_2$ that satisfies, for all $(q, r) \in R$ and $a \in A$,

- if $q \xrightarrow{a}_1 q'$ and $a \in X$ then there exists a $r' \in Q_2$ s.t. $r \xrightarrow{a}_2 r'$ and $(q', r') \in R$, and
- if $r \xrightarrow{a}_2 r'$ and $a \in Y$ then there exists a $q' \in Q_1$ s.t. $q \xrightarrow{a}_1 q'$ and $(q', r') \in R$.

We write $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ if there exists an XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$ that contains (q_1^0, q_2^0). Since the union of XY-simulations is an XY-simulation, $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ implies that there exists a unique maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. AA-simulations are just *bisimulations* [72], $A\emptyset$-simulations are *(forward) simulations* [69], and OI-simulations are *alternating simulations* [5]. AI-simulations appear in [3]. We write $\mathcal{I}_1 \sim \mathcal{I}_2$ instead of $\mathcal{I}_1 \sim_{AA} \mathcal{I}_2$.

The first lemma, which is trivial, states some basic transitivity, inclusion and symmetry properties of XY-simulations.

Lemma 7.1.3 *Suppose $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$ are IAs with inputs I and outputs O, $X \subseteq V \subseteq A$ and $Y \subseteq W \subseteq A = I \cup O$. Then*

1. *$\mathcal{I}_1 \sim_{XW} \mathcal{I}_2$ and $\mathcal{I}_2 \sim_{VY} \mathcal{I}_3$ implies $\mathcal{I}_1 \sim_{XY} \mathcal{I}_3$.*
2. *$\mathcal{I}_1 \sim_{VW} \mathcal{I}_2$ implies $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$.*
3. *$\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ implies $\mathcal{I}_2 \sim_{YX} \mathcal{I}_1$.*

The next technical lemma is the big work horse in this chapter.

Lemma 7.1.4 *Suppose $\mathcal{I}_1$ and $\mathcal{I}_2$ are IAs with $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$. Let R be the maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. Let $q_1, q_2 \in Q_1$ and $q_3, q_4 \in Q_2$, where Q_1 and Q_2 are the state sets of $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Then $q_1 \sim q_2 \wedge q_2\ R\ q_3 \wedge q_3 \sim q_4 \Rightarrow q_1\ R\ q_4$.*

Proof Let $R' = \{(q_1, q_4) \mid \exists q_2, q_3 : q_1 \sim q_2 \wedge q_2\ R\ q_3 \wedge q_3 \sim q_4\}$. It is routine to prove that R' is an XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. Since R is the maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$, $R' \subseteq R$. Now suppose $q_1 \sim q_2 \wedge q_2\ R\ q_3 \wedge q_3 \sim q_4$. By definition, $(q_1, q_4) \in R'$. Hence $(q_1, q_4) \in R$, as required.

The following lemma is used to link alternating simulations and the **ioco** relation.

Lemma 7.1.5 *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be determinate IAs such that $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$. Assume that $X \cup Y = A$, where A is the set of all (input and output) actions. Let R be the maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. Let $\sigma \in A^*$, $q_1 \in Q_1$ and $q_2 \in Q_2$ such that $q_1^0 \xrightarrow{\sigma}_* q_1$ and $q_2^0 \xrightarrow{\sigma}_* q_2$. Then $(q_1, q_2) \in R$.*

Proof By induction on the length of σ.

If $\mid \sigma \mid = 0$ then $\sigma = \epsilon$, $q_1 = q_1^0$ and $q_2 = q_2^0$. Since $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$ and R is the maximal XY-simulation, $(q_1^0, q_2^0) \in R$. Hence $(q_1, q_2) \in R$.

Now suppose $\mid \sigma \mid > 0$. Then there exist $\rho \in A^*$ and $a \in A$ such that $\sigma = \rho a$. Hence there exists states $q_1' \in Q_1$ and $q_2' \in Q_2$ such that $q_1^0 \xrightarrow{\rho}_* q_1' \xrightarrow{a} q_1$ and $q_2^0 \xrightarrow{\rho}_* q_2' \xrightarrow{a} q_2$. By induction hypothesis, $(q_1', q_2') \in R$. Since $X \cup Y = A$, either $a \in X$ or $a \in Y$. We consider two cases:

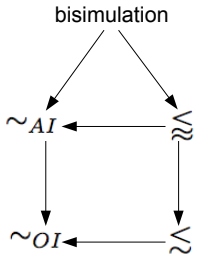

Figure 7.1: Relations Hierarchy

- $a \in X$. Since $q_1' \xrightarrow{a} q_1$, $(q_1', q_2') \in R$ and R is an XY-simulation, there exists a q_2'' such that $q_2' \xrightarrow{a} q_2''$ and $(q_1, q_2'') \in R$. Since $\mathcal{I}_2$ is determinate, $q_2'' \sim q_2$ and thus $(q_1, q_2) \in R$, by Lemma 7.1.4.
- $a \in Y$. Since $q_2' \xrightarrow{a} q_2$, $(q_1', q_2') \in R$ and R is an XY-simulation, there exists a q_1'' such that $q_1' \xrightarrow{a} q_1''$ and $(q_1'', q_2) \in R$. Since $\mathcal{I}_1$ is determinate, $q_1 \sim q_1''$ and thus $(q_1, q_2) \in R$, by Lemma 7.1.4.

7.1.4 Relating alternating simulations and ioco

The results below link alternating simulation and the **ioco** relation. Variations of these results occur in [3, 98].

Definition Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be IAs with inputs I and outputs O, and let $A = I \cup O$ and $A^\delta = A \cup \{\delta\}$. Then $\mathcal{I}_1 \lesssim \mathcal{I}_2 \;\Leftrightarrow\; \mathcal{I}_1^\delta \sim_{O^\delta I} \mathcal{I}_2^\delta$ and $\mathcal{I}_1 \lessapprox \mathcal{I}_2 \;\Leftrightarrow\; \mathcal{I}_1^\delta \sim_{A^\delta I} \mathcal{I}_2^\delta$.

In general, $\mathcal{I}_1 \lesssim \mathcal{I}_2$ implies $\mathcal{I}_1 \sim_{OI} \mathcal{I}_2$, but the converse implication does not hold. Similarly, $\mathcal{I}_1 \lessapprox \mathcal{I}_2$ implies $\mathcal{I}_1 \sim_{AI} \mathcal{I}_2$, but not vice versa.

Figure 7.1 depicts the hierarchy of these relations.

Lemma 7.1.6 *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be determinate IAs. Then $\mathcal{I}_1 \lesssim \mathcal{I}_2$ implies $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$.*

Proof Suppose that $\mathcal{I}_1 \lesssim \mathcal{I}_2$. Let $\sigma \in \mathit{Traces}(\mathcal{I}_2^\delta)$ and $o \in \mathit{out}(\mathcal{I}_1^\delta \textbf{ after } \sigma)$. We must prove $o \in \mathit{out}(\mathcal{I}_2^\delta \textbf{ after } \sigma)$. By the definitions, there exists $q_1 \in Q_1$ and $q_2 \in Q_2$ such that $q_1^0 \xrightarrow{\sigma}_* q_1$, $q_1 \xrightarrow{o}$ and $q_2^0 \xrightarrow{\sigma}_* q_2$. Let R be the maximal alternating simulation from $\mathcal{I}_1^\delta$ to $\mathcal{I}_2^\delta$. Since both $\mathcal{I}_1$ and $\mathcal{I}_2$ are determinate, $\mathcal{I}_1^\delta$ and $\mathcal{I}_2^\delta$ are determinate, by Lemma 7.1.1. Hence we can use Lemma 7.1.5 to obtain $(q_1, q_2) \in R$. It follows that $q_2 \xrightarrow{o}$, and hence $o \in \mathit{out}(\mathcal{I}_2^\delta \textbf{ after } \sigma)$, as required.

Lemma 7.1.7 *Let $\mathcal{I}_1$ be an IOA and let $\mathcal{I}_2$ be a determinate IA. Then $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$ implies $\mathcal{I}_1 \lesssim \mathcal{I}_2$.*

Proof Suppose $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$. Let $\mathcal{I}_1 = \langle I, O, Q_1, q_1^0, \rightarrow_1 \rangle$ and $\mathcal{I}_2 = \langle I, O, Q_2, q_2^0, \rightarrow_2 \rangle$. Define

$$R \;=\; \{(q_1, q_2) \in Q_1 \times Q_2 \mid \exists \sigma \in (I \cup O^\delta)^* : q_1^0 \xrightarrow{\sigma}_{1*} q_1 \wedge q_2^0 \xrightarrow{\sigma}_{2*} q_2\}.$$

We claim that R is an alternating simulation relation from $\mathcal{I}_1^\delta$ to $\mathcal{I}_2^\delta$.

Suppose that $(q_1, q_2) \in R$ and $q_1 \xrightarrow{o} q_1'$, for some $o \in O^\delta$. Then there exists a $\sigma \in (I \cup O^\delta)^*$ such that $q_1^0 \xrightarrow{\sigma}_{1*} q_1$ and $q_2^0 \xrightarrow{\sigma}_{2*} q_2$. Thus $\sigma \in \mathit{Traces}(\mathcal{I}_2^\delta)$ and $o \in \mathit{out}(\mathcal{I}_1^\delta \textbf{ after } \sigma)$. Using that $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$, we obtain $o \in \mathit{out}(\mathcal{I}_2^\delta \textbf{ after } \sigma)$. This means that there exists a state q_3 such that $q_2^0 \xrightarrow{\sigma}_{2*} q_3$ and $q_3 \xrightarrow{o}_2$. Since $\mathcal{I}_2$ is determinate, $\mathcal{I}_2^\delta$ is also determinate, by Lemma 7.1.1. Hence, by Lemma 7.1.2, $q_2 \sim q_3$. Hence there exists a state q_2' such that $q_2 \xrightarrow{o}_2 q_2'$. By definition of R, $(q_1', q_2') \in R$.

Now suppose that $(q_1, q_2) \in R$ and $q_2 \xrightarrow{i}_2 q_2'$, for some $i \in I$. As $\mathcal{I}_1$ is input-enabled, there exists a state q_1' such that $q_1 \xrightarrow{i}_1 q_1'$. By definition of R, $(q_1', q_2') \in R$.

By taking $\sigma = \epsilon$ in the definition of R, we obtain $(q_1^0, q_2^0) \in R$. Hence $\mathcal{I}_1 \lesssim \mathcal{I}_2$, as required.

7.2 Basic Framework for Inference of Automata

This section presents (a slight generalization of) the framework of [3] for learning interface automata. We assume there is a *teacher*, who knows a determinate IA $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, called the *system under test (SUT)*. There is also a *learner*, who has the task to learn about the behavior of $\mathcal{T}$ through experiments. The type of experiments which the learner may do is restricted by a *learning purpose* [3, 87, 99, 51], which is a determinate IA $\mathcal{P} = \langle I, O^\delta, P, p^0, \rightarrow_{\mathcal{P}} \rangle$, satisfying $\mathcal{T} \lesssim \mathcal{P}$.

In practice, there are various ways to ensure that $\mathcal{T} \lesssim \mathcal{P}$. If $\mathcal{T}$ is an IOA then $\mathcal{T} \lesssim \mathcal{P}$ is equivalent to $\mathcal{T}$ **ioco** $\mathcal{P}$ by Lemmas 7.1.6 and 7.1.7, and so we may use model-based black-box testing to obtain evidence for $\mathcal{T} \lesssim \mathcal{P}$. Alternatively, if $\mathcal{T}$ is an IOA and $\mathcal{P}$ is output-enabled then $\mathcal{T} \lesssim \mathcal{P}$ trivially holds.

After doing a number of experiments, the learner may formulate a *hypothesis*, which is a determinate IA $\mathcal{H}$ with outputs O^δ satisfying $\mathcal{H} \lessapprox \mathcal{P}$. Informally, the requirement $\mathcal{H} \lessapprox \mathcal{P}$ expresses that $\mathcal{H}$ only displays behaviors that are allowed by $\mathcal{P}$, but that any input action that must be explored according to $\mathcal{P}$ is indeed present in $\mathcal{H}$. Hypothesis $\mathcal{H}$ is *correct* if $\mathcal{T}$ **ioco** $\mathcal{H}$. In practice, we will use black-box testing to obtain evidence for the correctness of the hypothesis. In general, there will be many $\mathcal{H}$'s satisfying $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ (for instance, we may take $\mathcal{H} = \mathcal{P}$), and additional conditions will be imposed on $\mathcal{H}$, such as behavior-determinacy. In fact, section 7.2.1 establishes that if $\mathcal{T}$ is behavior-determininistic there always exists a behavior-deterministic IA $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$. If, in addition, $\mathcal{T}$ is an IOA then this $\mathcal{H}$ is unique up to bisimulation equivalence.

Learning purpose A trivial learning purpose $\mathcal{P}_{triv}$ is displayed in Figure 7.2 (left). Here notation $i : I$ means that we have an instance of the transition for each input $i \in I$. Notation $o : O$ is defined similarly. Since $\mathcal{P}_{triv}$ is output-enabled, $\mathcal{T} \lesssim \mathcal{P}_{triv}$ holds for each IOA $\mathcal{T}$. If $\mathcal{H}$ is a hypothesis, then $\mathcal{H} \lessapprox \mathcal{P}_{triv}$ just means that $\mathcal{H}$ is input enabled.

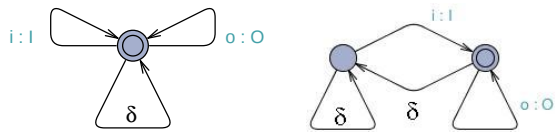

Figure 7.2: A trivial learning purpose (left) and a learning purpose with a nontrivial δ-transition (right).

The learning purpose $\mathcal{P}_{wait}$ displayed in Figure 7.2 (right) contains a nontrivial δ-transition. It expresses that after each input the learner has to wait until the SUT enters a quiescent state before offering the next input. It is straightforward to check that $\mathcal{T} \lesssim \mathcal{P}_{wait}$ holds if $\mathcal{T}$ is an IOA.

We now present the protocol that learner and teacher must follow. At any time, the teacher records the current state of $\mathcal{T}$, initially q^0, and the learner records the current state of $\mathcal{P}$, initially p^0. Suppose the teacher is in state q and the learner is in state p. In order to learn about the behavior of $\mathcal{T}$, the learner may engage in four types of interactions with the teacher:

1. *Input.* If a transition $p \xrightarrow{i}_{\mathcal{P}} p'$ is enabled in $\mathcal{P}$, then the learner may present input i to the teacher. If i is enabled in q then the teacher jumps to a state q' with $q \xrightarrow{i} q'$ and returns reply $\top$ to the learner. Otherwise, the teacher returns reply $\bot$. If the learner receives reply $\top$ it jumps to p', otherwise it stays in p.

2. *Output.* The learner may send an *output query* Δ to the teacher. Now there are two possibilities. If state q is quiescent, the teacher remains in q and returns answer δ. Otherwise, the teacher selects an output transition $q \xrightarrow{o} q'$, jumps to q', and returns o. The learner jumps to a state p' that can be reached by the answer o or δ.

3. *Reset.* The learner may send a **reset** to the teacher. In this case, both learner and teacher return to their respective initial states.

4. *Hypothesis.* The learner may present a *hypothesis* to the teacher: a determinate IA $\mathcal{H}$ with outputs O^δ such that $\mathcal{H} \lessapprox \mathcal{P}$. If $\mathcal{T}$ **ioco** $\mathcal{H}$ then the teacher returns answer **yes**. Otherwise, by definition, $\mathcal{H}^\delta$ has a trace σ such that an output o that is enabled by $\mathcal{T}^\delta$ **after** σ, is not enabled by $\mathcal{H}^\delta$ **after** σ. In this case, the teacher returns answer **no** together with counterexample σo, and learner and teacher return to their respective initial states.

The next lemma, which is easy to prove, implies that the teacher never returns $\bot$ to the learner: whenever the learner performs an input transition $p \xrightarrow{i}_{\mathcal{P}} p'$, the teacher can perform a matching transition $q \xrightarrow{i} q'$. Moreover, whenever the

teacher performs an output transition $q \xrightarrow{o} q'$, the learner can perform a matching transition $p \xrightarrow{o}_{\mathcal{P}} p'$.

Lemma 7.2.1 *Let R be the maximal alternating simulation from $\mathcal{T}^\delta$ to $\mathcal{P}^\delta$. Then, for any configuration of states q and p of teacher and learner, respectively, that can be reached after a finite number of steps (1)-(4) of the learning protocol, we have $(q, p) \in R$.*

Proof Routine, by induction on the number of steps, using Lemma 7.1.4 and the assumption that both $\mathcal{T}$ and $\mathcal{P}$ are determinate.

We are interested in effective procedures which, for any finite (and some infinite) $\mathcal{T}$ and $\mathcal{P}$ satisfying the above conditions, allow a learner to come up with a correct, behavior-deterministic hypothesis $\mathcal{H}$ after a finite number of interactions with the teacher. In [3], it is shown that any algorithm for learning Mealy machines can be transformed into an algorithm for learning finite, behavior-deterministic IOAs. Efficient algorithms for learning Mealy machines have been implemented in the tool Learnlib [83].

7.2.1 Existence and Uniqueness of Correct Hypothesis

In this section, it is established that if $\mathcal{T}$ is a behavior deterministic IOA and $\mathcal{P}$ is a determinate IA with $\mathcal{T} \lesssim \mathcal{P}$, there exists a unique behavior-deterministic IA $\mathcal{H}$ (up to bisimulation) such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$.

Lemma 7.2.2 *Suppose $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{I}_3$ and $\mathcal{I}_4$ are determinate IAs with the same sets I and O of inputs and outputs, respectively, such that $\mathcal{I}_1$ is active, and $\mathcal{I}_3$ and $\mathcal{I}_4$ are output-determined. Then $\mathcal{I}_1 \sim_{OI} \mathcal{I}_3 \sim_{AI} \mathcal{I}_2$ and $\mathcal{I}_1 \sim_{OI} \mathcal{I}_4 \sim_{AI} \mathcal{I}_2$ implies $\mathcal{I}_3 \sim \mathcal{I}_4$.*

Proof Let R_1 be the maximal alternating simulation from $\mathcal{I}_1$ to $\mathcal{I}_3$, R_2 be the maximal alternating simulation from $\mathcal{I}_1$ to $\mathcal{I}_4$, S_1 be the maximal AI-simulation from $\mathcal{I}_3$ to $\mathcal{I}_2$, S_2 be the maximal AI-simulation from $\mathcal{I}_4$ to $\mathcal{I}_2$, and let R be the relation between states of $\mathcal{I}_3$ and $\mathcal{I}_4$ given by:

$$\begin{aligned}(q_3, q_4) \in R \quad \Leftrightarrow \quad \exists q_1, q_2 : \quad & (q_1, q_3) \in R_1 \wedge \\ & (q_3, q_2) \in S_1 \wedge \\ & (q_1, q_4) \in R_2 \wedge \\ & (q_4, q_2) \in S_2.\end{aligned}$$

We claim that R is a bisimulation (AA-simulation) from $\mathcal{I}_3$ to $\mathcal{I}_4$.

Suppose $(q_3, q_4) \in R$ and $q_3 \xrightarrow{a} q_3'$. The there exist q_1 and q_2 such that $(q_1, q_3) \in R_1$, $(q_3, q_2) \in S_1$, $(q_1, q_4) \in R_2$, and $(q_4, q_2) \in S_2$. We consider two cases:

- $a \in I$. Since S_1 is an AI-simulation, there exists a state q_2' such that $q_2 \xrightarrow{a} q_2'$ and $(q_3', q_2') \in S_1$. Since S_2 is an AI-simulation, there exists a state q_4' such that $q_4 \xrightarrow{a} q_4'$ and $(q_4', q_2') \in S_2$. Since R_2 is an OI-simulation, there exists a state q_1' such that $q_1 \xrightarrow{a} q_1'$ and $(q_1', q_4') \in R_2$. Since R_1 is an OI-simulation, there exists a state q_1'' such that $q_1 \xrightarrow{a} q_1''$ and $(q_1'', q_3') \in R_1$. Since $\mathcal{I}_1$ is determinate, $q_1' \sim q_1''$. Combination of $q_1' \sim q_1''$ and $(q_1'', q_3') \in R_1$ gives $(q_1', q_3') \in R_1$, using Lemma 7.1.4 and the assumption that R_1 is maximal. Hence $(q_3', q_4') \in R$, by definition of R.

- $a \in O$. Since $\mathcal{I}_1$ is active, there exists a transition $q_1 \xrightarrow{o} q_1'$, for some output o. Since R_1 is an OI-simulation, there exists a state q_3'' such that $q_3 \xrightarrow{o} q_3''$ and $(q_1', q_3'') \in R_1$. Since $\mathcal{I}_3$ is behavior-deterministic, $o = a$ and $q_3'' \sim q_3'$. Hence $(q_1', q_3') \in R_1$. Since R_2 is an OI-simulation, there exists a state q_4' such that $q_4 \xrightarrow{a} q_4'$ and $(q_1', q_4') \in R_2$. Since S_2 is an AI-simulation, there exists a state q_2' such that $q_2 \xrightarrow{a} q_2'$ and $(q_4', q_2') \in S_2$. Since S_1 is an AI-simulation, there exists a state q_2'' such that $q_2 \xrightarrow{a} q_2''$ and $(q_3', q_2'') \in S_1$. Since $\mathcal{I}_2$ is determinate, $q_2'' \sim q_2'$. Combination of $(q_3', q_2'') \in S_1$ and $q_2'' \sim q_2'$ gives $(q_3', q_2') \in S_1$, using Lemma 7.1.4 and the assumption that S_1 is maximal. Hence $(q_3', q_4') \in R$, by definition of R.

The proof of the case that $(q_3, q_4) \in R$ and $q_4 \xrightarrow{a} q_4'$ is fully symmetric.

It is immediate from the definitions that $(q_3^0, q_4^0) \in R$. Hence $\mathcal{I}_3 \sim \mathcal{I}_4$, as required.

Suppose that $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$. $XY(\mathcal{I}_1, \mathcal{I}_2)$ is defined to be the product interface automaton induced by $\sim_{XY}$, as the structure $\langle I, O, R, (q_1^0, q_2^0), \rightarrow \rangle$ where R is the maximal XY-simulation relation from $\mathcal{I}_1$ to $\mathcal{I}_2$ and $(q, r) \xrightarrow{a} (q', r') \Leftrightarrow q \xrightarrow{a}_1 q' \wedge r \xrightarrow{a}_2 r'$.

Lemma 7.2.3 *Suppose that $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$. Then $\mathcal{I}_1 \sim_{XA} XY(\mathcal{I}_1, \mathcal{I}_2) \sim_{AY} \mathcal{I}_2$.*

Proof Let R be the maximal XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$. Let $R_1 = \{(q, (q, r)) \mid (q, r) \in R\}$ and $R_2 = \{((q, r), r)) \mid (q, r) \in R\}$. It is straightforward to check that R_1 is an XA-simulation from $\mathcal{I}_1$ to $XY(\mathcal{I}_1, \mathcal{I}_2)$, and r_2 is an AY-simulation from $XY(\mathcal{I}_1, \mathcal{I}_2)$ to $\mathcal{I}_2$. Since R is an XY-simulation from $\mathcal{I}_1$ to $\mathcal{I}_2$, it contains the pair (q_1^0, q_2^0). Hence R_1 contains $(q_1^0, (q_1^0, q_2^0))$ and R_2 contains $((q_1^0, q_2^0), q_2^0)$, so the initial states are related, as required.

Lemma 7.2.4 *Suppose that $\mathcal{I}_1 \sim_{XY} \mathcal{I}_2$.*

1. *If $\mathcal{I}_1$ and $\mathcal{I}_2$ are determinate, then $XY(\mathcal{I}_1, \mathcal{I}_2)$ is determinate.*
2. *If $\mathcal{I}_1$ or $\mathcal{I}_2$ is output-determined, then $XY(\mathcal{I}_1, \mathcal{I}_2)$ is output-determined.*

The following theorem proves that for a given behavior-deterministic SUT and a determinate learning purpose, learner will necessarily find a behavior-deterministic learning hypothesis.

Theorem 7.2.5 *Suppose $\mathcal{T}$ is a behavior-deterministic IA and $\mathcal{P}$ is a determinate IA such that $\mathcal{T} \lesssim \mathcal{P}$. Then there exists a behavior-deterministic IA $\mathcal{H}$ such that $\mathcal{T}$* **ioco** *$\mathcal{H} \lessapprox \mathcal{P}$.*

Proof By expanding the definition of $\lesssim$, we obtain $\mathcal{T}^\delta \sim_{O^\delta I} \mathcal{P}^\delta$. Thus, by Lemma 7.2.3,

$$\mathcal{T}^\delta \sim_{O^\delta A^\delta} O^\delta I(\mathcal{T}^\delta, \mathcal{P}^\delta) \sim_{A^\delta I} \mathcal{P}^\delta.$$

Let $\mathcal{H} = O^\delta I(\mathcal{T}^\delta, \mathcal{P}^\delta)$. Since $\mathcal{H}$ is active, $\mathcal{H}^\delta = \mathcal{H}$. Hence, by Lemma 7.1.3(2),

$$\mathcal{T}^\delta \sim_{O^\delta I} \mathcal{H}^\delta \sim_{A^\delta I} \mathcal{P}^\delta.$$

By the definitions of $\lesssim$ and $\lessapprox$, we obtain $\mathcal{T} \lesssim \mathcal{H} \lessapprox \mathcal{P}$. Lemma 7.1.1 and Lemma 7.2.4 imply that $\mathcal{H}$ is behavior-deterministic, as required. By Lemma 7.1.6, $\mathcal{T}$ **ioco** $\mathcal{H}$.

Next theorem, proves that for a given behavior-deterministic SUT and a determinate learning purpose, learning hypothesis is unique.

Theorem 7.2.6 *Let $\mathcal{T}$ be a behavior-deterministic IOA over I and O, $\mathcal{H}_1$ and $\mathcal{H}_2$ behavior-deterministic IAs over I and O^δ, and $\mathcal{P}$ a determinate IA over I and O^δ such that $\mathcal{T}$* **ioco** *$\mathcal{H}_1 \lessapprox \mathcal{P}$ and $\mathcal{T}$* **ioco** *$\mathcal{H}_2 \lessapprox \mathcal{P}$. Then $\mathcal{H}_1^\delta \sim \mathcal{H}_2^\delta$.*

Proof Since $\mathcal{T}$ is an IOA, $\mathcal{H}_1$ is determinate and $\mathcal{T}$ **ioco** $\mathcal{H}_1$, it follows by Lemma 7.1.7 that $\mathcal{T} \lesssim \mathcal{H}_1$. Similarly, we derive $\mathcal{T} \lesssim \mathcal{H}_2$. By expanding the definitions of $\lesssim$ and $\lessapprox$, we obtain:

$$\mathcal{T}^\delta \sim_{O^\delta I} \mathcal{H}_1^\delta \sim_{A^\delta I} \mathcal{P}^\delta \text{ and } \mathcal{T}^\delta \sim_{O^\delta I} \mathcal{H}_2^\delta \sim_{A^\delta I} \mathcal{P}^\delta$$

From the assumptions and Lemma 7.1.1, it follows that $\mathcal{T}^\delta$, $\mathcal{H}_1^\delta$, $\mathcal{H}_2^\delta$ and $\mathcal{P}^\delta$ are determinate, $\mathcal{T}^\delta$ is active, and $\mathcal{H}_1^\delta$ and $\mathcal{H}_2^\delta$ are output-determined. Hence we may apply Lemma 7.2.2 to obtain $\mathcal{H}_1^\delta \sim \mathcal{H}_2^\delta$.

7.3 Mappers

In order to learn a "large" IA $\mathcal{T}$, with inputs I and outputs O, a *mapper* is placed between the teacher and the learner, which translates concrete actions in I and O to abstract actions in (typically smaller) sets X and Y, and vice versa. The task of the learner is then reduced to inferring a "small" IA with alphabet X and Y. Our notion of mapper is essentially the same as the one of [1].

Mapper A *mapper* for a set of inputs I and a set of outputs O is a tuple $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$, where

- $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$ is a deterministic IA that is input- and output-enabled and has trivial δ-transitions: $r \xrightarrow{\delta} r' \Leftrightarrow r = r'$.

- X and Y are disjoint sets of abstract input and output actions with $\delta \in Y$.
- $\Upsilon : R \times A^\delta \to Z$, where $A = I \cup O$ and $Z = X \cup Y$, maps concrete actions to abstract ones. We write $\Upsilon_r(a)$ for $\Upsilon(r, a)$ and require that Υ_r respects inputs, outputs and quiescence:

$$(\Upsilon_r(a) \in X \Leftrightarrow a \in I) \wedge (\Upsilon_r(a) = \delta \Leftrightarrow a = \delta).$$

Mapper $\mathcal{A}$ is *output-predicting* if $\forall o, o' \in O : \Upsilon_r(o) = \Upsilon_r(o') \Rightarrow o = o'$, that is, Υ_r is injective on outputs, for each $r \in R$. Mapper $\mathcal{A}$ is *surjective* if $\forall z \in Z\ \exists a \in A^\delta : \Upsilon_r(a) = z$, that is, Υ_r is surjective, for each $r \in R$. Mapper $\mathcal{A}$ is *state-free* if R is a singleton set.

Example Consider a system with input actions $LOGIN(p_1)$, $SET(p_2)$ and $LOGOUT$. Assume that the system only triggers certain outputs when a user is properly logged in. Then we may not abstract from the password parameters p_1 and p_2 entirely, since this will lead to nondeterminism. We may preserve behavior-determinism by considering just two abstract values for p_1: ok and nok. Since passwords can be changed using the input $SET(p_2)$ when a user is logged in, the mapper may not be state-free: it has to record the current password and whether or not the user is logged (T and F, respectively). The input transitions are defined by:

$$\begin{array}{rl} & (p,b) \xrightarrow{LOGIN(p)} (p,\mathsf{T}) \\ p \neq p_1 \Rightarrow & (p,b) \xrightarrow{LOGIN(p_1)} (p,b) \\ & (p,\mathsf{T}) \xrightarrow{SET(p_2)} (p_2,\mathsf{T}) \\ & (p,\mathsf{F}) \xrightarrow{SET(p_2)} (p,\mathsf{F}) \\ & (p,b) \xrightarrow{LOGOUT} (p,\mathsf{F}) \end{array}$$

For input actions, abstraction Υ is defined by

$$\begin{array}{rcl} \Upsilon_{(p,b)}(LOGIN(p_1)) & = & \begin{cases} LOGIN(\mathsf{ok}) & \text{if } p_1 = p \\ LOGIN(\mathsf{nok}) & \text{otherwise} \end{cases} \\ \Upsilon_{(p,b)}(SET(p_2)) & = & SET \end{array}$$

For input $LOGOUT$ and for output actions, $\Upsilon_{(p,b)}$ is the identity. This mapper is surjective, since no matter how the password has been set, a user may always choose either a correct or an incorrect login.

Example Consider a system with three inputs $IN1(n_1)$, $IN2(n_2)$, and $IN3(n_3)$, in which an $IN3(n_3)$ input triggers an output OK if and only if the value of n_3 equals either the latest value of n_1 or the latest value of n_2. In this case, we may not abstract away entirely from the values of the parameters, since that leads to nondeterminism. We may preserve behavior-determinism by a mapper that records the last values of n_1 and n_2. Thus, if D is the set of parameter values, the

set of mapper states is defined by $R = (D \cup \{\bot\}) \times (D \cup \{\bot\})$, choose $r^0 = (\bot, \bot)$ as initial state, and define the input transitions by

$$(v_1, v_2) \xrightarrow{IN1(n_1)} (n_1, v_2)$$
$$(v_1, v_2) \xrightarrow{IN2(n_2)} (v_1, n_2)$$
$$(v_1, v_2) \xrightarrow{IN3(n_3)} (v_1, v_2)$$

Abstraction Υ abstracts from the specific value of a parameter, and only records whether it is fresh, or equals the last value of *IN1* or *IN2*. For $i = 1, 2, 3$:

$$\Upsilon_{(v_1,v_2)}(INi(n_i)) \quad = \quad \begin{cases} INi(\mathsf{old}_1) & \text{if } n_i = v_1 \\ INi(\mathsf{old}_2) & \text{if } n_i = v_2 \wedge n_i \neq v_1 \\ INi(\mathsf{fresh}) & \text{otherwise} \end{cases}$$

This abstraction is not surjective: for instance, in the initial state $IN1(\mathsf{old}_1)$ is not possible as an abstract value, and in any state of the form (v, v), $IN1(\mathsf{old}_2)$ is not possible.

Each mapper $\mathcal{A}$ induces an abstraction operator on interface automata, which abstracts an IA with actions in I and O into an IA with actions in X and Y. This abstraction operator is essentially just a variation of the state operator well-known from process algebras [9].

Abstraction Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$ be an IA and let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper with $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$. Then $\alpha_{\mathcal{A}}(\mathcal{T})$, the *abstraction* of $\mathcal{T}$, is the IA $\langle X, Y, Q \times R, (q^0, r^0), \rightarrow_{\mathsf{abst}} \rangle$, where transition relation $\rightarrow_{\mathsf{abst}}$ is given by the rule:

$$\frac{q \xrightarrow{a} q' \qquad r \xrightarrow{a} r' \qquad \Upsilon_r(a) = z}{(q, r) \xrightarrow{z}_{\mathsf{abst}} (q', r')}$$

Observe that if $\mathcal{T}$ is determinate then $\alpha_{\mathcal{A}}(\mathcal{T})$ does not have to be determinate. Also, if $\mathcal{T}$ is an IOA then $\alpha_{\mathcal{A}}(\mathcal{T})$ does not have to be an IOA (if $\mathcal{A}$ is not surjective, as in Example 7.3, then an abstract input will not be enabled if there is no corresponding concrete input). If $\mathcal{T}$ is output-determined then $\alpha_{\mathcal{A}}(\mathcal{T})$ is output-determined, but the converse implication does not hold. The following lemma gives a positive result: abstraction is monotone with respect to the alternating simulation preorder.

Lemma 7.3.1 *If $\mathcal{T}_1 \lesssim \mathcal{T}_2$ then $\alpha_{\mathcal{A}}(\mathcal{T}_1) \lesssim \alpha_{\mathcal{A}}(\mathcal{T}_2)$.*

Proof Suppose $\mathcal{T}_1 \lesssim \mathcal{T}_2$. Let R be the maximal alternating simulation from $\mathcal{T}_1^\delta$ to $\mathcal{T}_2^\delta$. Define the relation R' between states of $\alpha_{\mathcal{A}}(\mathcal{T}_1)$ and $\alpha_{\mathcal{A}}(\mathcal{T}_2)$ as follows:

$$(q_1, r_1)\ R'\ (q_2, r_2) \quad \Leftrightarrow \quad q_1\ R\ q_2 \wedge r_1 = r_2.$$

It is routine to prove that R' is an alternating simulation from $(\alpha_{\mathcal{A}}(\mathcal{T}_1))^\delta$ to $(\alpha_{\mathcal{A}}(\mathcal{T}_2))^\delta$. Hence $\alpha_{\mathcal{A}}(\mathcal{T}_1) \lesssim \alpha_{\mathcal{A}}(\mathcal{T}_2)$, as required.

The concretization operator is the dual of the abstraction operator. It transforms each IA with abstract actions in X and Y into an IA with concrete actions in I and O.

Concretization Let $\mathcal{H} = \langle X, Y, S, s^0, \rightarrow \rangle$ be an IA and let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper for I and O. Then $\gamma_{\mathcal{A}}(\mathcal{H})$, the *concretization* of $\mathcal{H}$, is the IA $\langle I, O^{\delta}, R \times S, (r^0, s^0), \rightarrow_{\mathsf{conc}} \rangle$, where transition relation $\rightarrow_{\mathsf{conc}}$ is given by the rule:

$$\frac{r \xrightarrow{a} r' \qquad s \xrightarrow{z} s' \qquad \Upsilon_r(a) = z}{(r, s) \xrightarrow{a}_{\mathsf{conc}} (r', s')}$$

Whereas the abstraction operator does not preserve determinacy in general, the concretization of a determinate IA is always determinate. Also, the concretization of an output-determined IA is output-determined, provided the mapper is output-predicting.

Lemma 7.3.2 *If $\mathcal{H}$ is determinate then $\gamma_{\mathcal{A}}(\mathcal{H})$ is determinate.*

Proof Routine. It is easy to show that the relation $R = \{(r, s), (r, s') \mid s \sim s'\}$ is a bisimulation on $\gamma_{\mathcal{A}}(\mathcal{H})$. Now suppose that (r, s) is a reachable state of $\gamma_{\mathcal{A}}(\mathcal{H})$ with outgoing transitions $(r, s) \xrightarrow{a}_{\mathsf{conc}} (r_1, s_1)$ and $(r, s) \xrightarrow{a}_{\mathsf{conc}} (r_2, s_2)$. Then, by definition of $\gamma_{\mathcal{A}}(\mathcal{H})$, $r \xrightarrow{a} r_1$, $s \xrightarrow{z} s_1$, where $z = \Upsilon_r(a)$, $r \xrightarrow{a} r_2$ and $s \xrightarrow{z} s_2$. Since the IA of $\mathcal{A}$ is deterministic, $r_1 = r_2$. Since (r, s) is reachable in $\gamma_{\mathcal{A}}(\mathcal{H})$, s is reachable in $\mathcal{H}$. Hence, because $\mathcal{H}$ is determinate, $s_1 \sim s_2$. It follows that $((r_1, s_1), (r_2, s_2)) \in R$. Since R is a bisimulation, we conclude $(r_1, s_1) \sim (r_2, s_2)$, as required.

Lemma 7.3.3 *If $\mathcal{A}$ is output-predicting and $\mathcal{H}$ is output-determined then $\gamma_{\mathcal{A}}(\mathcal{H})$ is output-determined.*

Proof Suppose that $\mathcal{A}$ is output-predicting and $\mathcal{H}$ is output-determined. Let (r, s) be a reachable state of $\gamma_{\mathcal{A}}(\mathcal{H})$ such that, for concrete outputs o and o', $(r, s) \xrightarrow{o}_{\mathsf{conc}}$ and $(r, s) \xrightarrow{o'}_{\mathsf{conc}}$. Then it follows from the definition of $\gamma_{\mathcal{A}}(\mathcal{H})$ that there exists abstract outputs y and y' such that $s \xrightarrow{y}$, $s \xrightarrow{y'}$, $\Upsilon_r(o) = y$ and $\Upsilon_r(o') = y'$. Since (r, s) is reachable in $\gamma_{\mathcal{A}}(\mathcal{H})$, it follows that s is reachable in $\mathcal{H}$. Hence, by the assumption that $\mathcal{H}$ is output-determined, $y = y'$. Next, using that $\mathcal{A}$ is output-predicting, we conclude $o = o'$.

In an abstraction of the form $\gamma_{\mathcal{A}}(\mathcal{H})$ it may occur that a reachable state (r, s) is quiescent, even though the contained state s of $\mathcal{H}$ enables some abstract output y: this happens if there exists no concrete concrete output o such that $\Upsilon_r(o) = y$. This situation is ruled out by following definition.

Definition $\gamma_{\mathcal{A}}(\mathcal{H})$ is *quiescence preserving* if, for each reachable state (r, s), (r, s) quiescent implies s quiescent.

Concretization is monotone with respect to the $\lessapprox$ preorder, provided the concretization of the first argument is quiescence preserving.

Lemma 7.3.4 *Suppose $\gamma_{\mathcal{A}}(\mathcal{H}_1)$ is quiescence preserving. Then $\mathcal{H}_1 \lessapprox \mathcal{H}_2$ implies $\gamma_{\mathcal{A}}(\mathcal{H}_1) \lessapprox \gamma_{\mathcal{A}}(\mathcal{H}_2)$.*

Proof Suppose $\mathcal{H}_1 \lessapprox \mathcal{H}_2$. Let R be the maximal $A^\delta I$-simulation from $\mathcal{H}_1^\delta$ to $\mathcal{H}_2^\delta$. Define relation R' between states of $\gamma_{\mathcal{A}}(\mathcal{H}_1)$ and $\gamma_{\mathcal{A}}(\mathcal{H}_2)$ as follows:

$$(r_1, s_1)\ R'\ (r_2, s_2) \quad \Leftrightarrow \quad r_1 = r_2 \wedge s_1\ R\ s_2.$$

We check that R' is an $A^\delta I$-simulation from $(\gamma_{\mathcal{A}}(\mathcal{H}_1))^\delta$ to $(\gamma_{\mathcal{A}}(\mathcal{H}_2))^\delta$. Suppose $(r, s_1)\ R'\ (r, s_2)$.

- Suppose $(r, s_2) \xrightarrow{i} (r', s_2')$ for some $i \in I$. Let $\Upsilon_r(i) = x$. Then, by definition of concretization, $r \xrightarrow{i} r'$ and $s_2 \xrightarrow{x} s_2'$. Using that $s_1\ R\ s_2$, we infer that there exists a state s_1' such that $s_1 \xrightarrow{x} s_1'$ and $s_1'\ R\ s_2'$. Hence $(r, s_1) \xrightarrow{i} (r', s_1')$ and $(r', s_1')\ R'\ (r', s_2')$, as required.
- Suppose $(r, s_1) \xrightarrow{a} (r', s_1')$ for some $a \in A^\delta$, $\Upsilon_r(a) = z$, $r \xrightarrow{a} r'$ and $s_1 \xrightarrow{z} s_1'$. Using that $s_1\ R\ s_2$, we infer that there exists a state s_2' such that $s_2 \xrightarrow{z} s_2'$ and $s_1'\ R\ s_2'$. Hence $(r, s_2) \xrightarrow{a} (r', s_2')$ and $(r', s_1')\ R'\ (r', s_2')$, as required.
- Suppose (r, s_1) is quiescent. Then, since $\gamma_{\mathcal{A}}(\mathcal{H}_1)$ is quiescence preserving, s_1 is quiescent. Since $s_1\ R\ s_2$ and R is a $A^\delta I$-simulation from $\mathcal{H}_1^\delta$ to $\mathcal{H}_2^\delta$, it follows that s_2 is quiescent. Hence, by definition of concretization, (r, s_2) is quiescent.

Since R is a $A^\delta I$-simulation from $\mathcal{H}_1^\delta$ to $\mathcal{H}_2^\delta$, $s_1^0\ R\ s_2^0$. Hence we have $(r^0, s_1^0)\ R'\ (r^0, s_2^0)$ and so R' relates the initial states of $\gamma_{\mathcal{A}}(\mathcal{H}_1)$ and $\gamma_{\mathcal{A}}(\mathcal{H}_2)$. Thus $\gamma_{\mathcal{A}}(\mathcal{H}_1) \lessapprox \gamma_{\mathcal{A}}(\mathcal{H}_2)$, as required.

The lemma below is a key result of this chapter. It says that if $\mathcal{T}$ is **ioco**-conforming to the concretization of an hypothesis $\mathcal{H}$, and this concretization is quiescence preserving, then the abstraction of $\mathcal{T}$ is **ioco**-conforming to $\mathcal{H}$ itself.

Lemma 7.3.5 *Suppose $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving. Then $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$ implies $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}$.*

Proof Suppose $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$. Let $\sigma \in \mathit{Traces}(\mathcal{H}^\delta)$ and let $y \in \mathit{out}((\alpha_{\mathcal{A}}(\mathcal{T}))^\delta$ **after** $\sigma)$. We must show that $y \in \mathit{out}(\mathcal{H}^\delta$ **after** $\sigma)$. Let $\sigma = z_1 \cdots z_n$. Then $\mathcal{H}^\delta$ has a run

$$s_0 \xrightarrow{z_1} s_1 \xrightarrow{z_2} \cdots \xrightarrow{z_n} s_n$$

with $s_0 = s^0$, and $(\alpha_{\mathcal{A}}(\mathcal{T}))^\delta$ has a run

$$(q_0, r_0) \xrightarrow{z_1} (q_1, r_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, r_n) \xrightarrow{y}$$

with $(q_0, r_0) = (q^0, r^0)$. Then, by definition of $(\alpha_{\mathcal{A}}(\mathcal{T}))^\delta$, there exists runs

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n \xrightarrow{o}$$

$$r_0 \xrightarrow{a_1} r_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} r_n \xrightarrow{o}$$

of $\mathcal{T}^\delta$ and $\mathcal{A}$, respectively, such that, for all $1 \leq i \leq n$, $\Upsilon_{r_{i-1}}(a_i) = z_i$ and $\Upsilon_{r_n}(o) = y$. By definition of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$, this IA has a run

$$(r_0, s_0) \xrightarrow{a_1} (r_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n)$$

Let $\rho = a_1 \cdots a_n$. Then $\rho \in \mathit{Traces}((\gamma_{\mathcal{A}}(\mathcal{H}))^\delta)$. Moreover, $o \in \mathit{out}(\mathcal{T}^\delta \textbf{ after } \rho)$. Using $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$, we obtain $o \in \mathit{out}((\gamma_{\mathcal{A}}(\mathcal{H}))^\delta \textbf{ after } \rho)$. Hence $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ has a run

$$(r_0, s_0') \xrightarrow{a_1} (r_1, s_1') \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n') \xrightarrow{o}$$

By definition of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ and using that $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescent preserving, we may infer that $\mathcal{H}^\delta$ has a run

$$s_0' \xrightarrow{z_1} s_1' \xrightarrow{z_2} \cdots \xrightarrow{z_n} s_n' \xrightarrow{y}$$

Hence, $y \in \mathit{out}(\mathcal{H}^\delta \textbf{ after } \sigma)$, as required.

By using a mapper $\mathcal{A}$, we may reduce the task of learning an IA $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ to the simpler task of learning an IA $\mathcal{H}'$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}' \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. However, in order to establish the correctness of this reduction, we need two technical lemmas that require some additional assumptions on $\mathcal{P}$ and $\mathcal{A}$. It is straightforward to check that these assumptions are met by the mappers of Examples 7.3 and 7.3, and the learning purposes of Example 7.2.

Definition Let $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ be a mapper for I and O. $\equiv_{\mathcal{A}}$ is defined to be the equivalence relation on $I \cup O^\delta$ which declares two concrete actions equivalent if, for some states of the mapper, they are mapped to the same abstract action: $a \equiv_{\mathcal{A}} b \;\Leftrightarrow\; \exists r, r' : \Upsilon_r(a) = \Upsilon_{r'}(b)$. Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$ be an IA. $\mathcal{P}$ and $\mathcal{A}$ are called *compatible* if, for all concrete actions a, b with $a \equiv_{\mathcal{A}} b$ and for all $p, p_1, p_2 \in P$, $(p \xrightarrow{a} \Leftrightarrow p \xrightarrow{b}) \wedge (p \xrightarrow{a} p_1 \wedge p \xrightarrow{b} p_2 \Rightarrow p_1 \sim p_2)$.

Lemma 7.3.6 *Suppose $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate and $\mathcal{P}$ and $\mathcal{A}$ are compatible. Then $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P})) \lessapprox \mathcal{P}$.*

Proof We claim $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P})) \sim \mathcal{P}$. In order to prove this, consider the relation

$$S \;=\; \{((r_2, (p_1, r_1)), p_2) \mid p_1 \sim p_2 \wedge (p_1, r_1) \sim (p_2, r_2)\}.$$

It is easy to check that S relates the initial states of $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P}))$ and $\mathcal{P}$. We show that S is a bisimulation.

Suppose $((r_2, (p_1, r_1)), p_2) \in S$ and $p_2 \xrightarrow{a} p_2'$. Since the IA for $\mathcal{A}$ is input- and output-enabled, there exist a state r_2' such that $r_2 \xrightarrow{a} r_2'$. Let $\Upsilon_{r_2}(a) = z$. Then, by

definition of the abstraction operator, $(p_2, r_2) \xrightarrow{z} (p_2', r_2')$. Since $(p_1, r_1) \sim (p_2, r_2)$, there exists a pair (p_1', r_1') such that $(p_1, r_1) \xrightarrow{z} (p_1', r_1')$ and $(p_1', r_1') \sim (p_2', r_2')$. By definition of the abstraction operator, there exists a concrete action b such that $\Upsilon_{r_1}(b) = z$, $p_1 \xrightarrow{b} p_1'$ and $r_1 \xrightarrow{b} r_1'$. Since $p_1 \sim p_2$, there exists a p_2'' such that $p_2 \xrightarrow{b} p_2''$ and $p_1' \sim p_2''$. Since $\mathcal{P}$ and $\mathcal{A}$ are compatible and $a \equiv_{\mathcal{A}} b$, $p_2' \sim p_2''$. By Lemma 7.1.4, $p_1' \sim p_2'$. By definition of the concretization operator, $(r_2, (p_1, r_1)) \xrightarrow{a} (r_2', (p_1', r_1'))$. Moreover, $((r_2', (p_1', r_1')), p_2') \in S$, as required.

Suppose $((r_2, (p_1, r_1)), p_2) \in S$ and $(r_2, (p_1, r_1)) \xrightarrow{a} (r_2', (p_1', r_1'))$. Let $\Upsilon_{r_2}(a) = z$. Then, by definition of the concretization operator, $r_2 \xrightarrow{a} r_2'$ and $(p_1, r_1) \xrightarrow{z} (p_1', r_1')$. By definition of the abstraction operator, there exists a concrete action b such that $\Upsilon_{r_1}(b) = z$, $p_1 \xrightarrow{b} p_1'$ and $r_1 \xrightarrow{b} r_1'$. Since $\mathcal{P}$ and $\mathcal{A}$ are compatible and $a \equiv_{\mathcal{A}} b$, there exists a p_1'' such that $p_1 \xrightarrow{a} p_1''$ and $p_1'' \sim p_1'$. Since $p_1 \sim p_2$, by Lemma 7.1.4 there exists a p_2' such that $p_2 \xrightarrow{a} p_2'$ and $p_1' \sim p_2'$. By definition of the abstraction operator, $(p_2, r_2) \xrightarrow{z} (p_2', r_2')$. Since $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate, it follows by Lemma 7.1.4 that $(p_1', r_1') \sim (p_2', r_2')$. Hence, $((r_2', (p_1', r_1')), p_2') \in S$, as required.

Now the lemma follows since, for all IA's $\mathcal{I}_1$ and $\mathcal{I}_2$, $\mathcal{I}_1 \sim \mathcal{I}_2 \Rightarrow \mathcal{I}_1^{\delta} \sim \mathcal{I}_2^{\delta} \Rightarrow \mathcal{I}_1 \lessapprox \mathcal{I}_2$.

Lemma 7.3.7 *Suppose $\mathcal{A}$ and $\mathcal{P}$ are compatible, $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate and $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. Then $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving.*

Proof By contradiction. Assume that $\gamma_{\mathcal{A}}(\mathcal{H})$ is not quiescence preserving. Consider a minimal run that shows this, that is, a run

$$(r_0, s_0) \xrightarrow{a_1} (r_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n) \xrightarrow{a_{n+1}}$$

of $(\gamma_{\mathcal{A}}(\mathcal{H}))^{\delta}$ with $(r_0, s_0) = (r^0, s^0)$, (r_n, s_n) quiescent in $\gamma_{\mathcal{A}}(\mathcal{H})$, so $a_{n+1} = \delta$, and s_n not quiescent in $\mathcal{H}$. Let $z_j = \Upsilon_{r_{j-1}}(a_j)$, for $1 \leq j \leq n$, and let $z_{n+1} \neq \delta$ be an output action enabled in state s_n of $\mathcal{H}$. Since (r_n, s_n) is quiescent, it follows that there exists no concrete output o such that $\Upsilon_{r_n}(o) = z_{n+1}$. From the definition of concretization and the minimality of the run of $(\gamma_{\mathcal{A}}(\mathcal{H}))^{\delta}$, it follows that

$$r_0 \xrightarrow{a_1} r_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} r_n \xrightarrow{a_{n+1}}$$

is a run of the IA of $\mathcal{A}$, and

$$s_0 \xrightarrow{z_1} s_1 \xrightarrow{z_2} \cdots \xrightarrow{z_n} s_n \xrightarrow{z_{n+1}}$$

is a run of $\mathcal{H}$. Let S be the maximal $A^{\delta}I$-simulation from $\mathcal{H}^{\delta}$ to $(\alpha_{\mathcal{A}}(\mathcal{P}))^{\delta}$. Then, using the assumptions that $\alpha_{\mathcal{A}}(\mathcal{P})$ is determinate and that $\mathcal{P}$ and $\mathcal{A}$ are compatible, we may construct runs

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n$$

and

$$(p_0, r_0) \xrightarrow{z_1} (p_1, r_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (p_n, r_n) \xrightarrow{z_{n+1}}$$

of $\mathcal{P}^{\delta}$ and $(\alpha_{\mathcal{A}}(\mathcal{P}))^{\delta}$, respectively, such that, for all $i \leq n$, $(s_i, (p_i, r_i)) \in S$. But since $(p_n, r_n) \xrightarrow{z_{n+1}}$, it follows that there exists a concrete output o such that $\Upsilon_{r_n}(o) = z_{n+1}$. Contradiction.

7.4 Inference Using Abstraction

Suppose we have a teacher equipped with a determinate IA $\mathcal{T}$, and a learner equipped with a determinate learning purpose $\mathcal{P}$ such that $\mathcal{T} \lessapprox \mathcal{P}$. The learner has the task to infer some $\mathcal{H}$ satisfying $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$. After the preparations from the previous section, we are now ready to show how, in certain cases, the learner may simplify her task by defining a mapper $\mathcal{A}$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ and $\alpha_{\mathcal{A}}(\mathcal{P})$ are determinate, $\mathcal{P}$ and $\mathcal{A}$ are compatible, and $\mathcal{T}$ *respects* $\mathcal{A}$ in the sense that, for $i, i' \in I$ and $q \in Q$, $i \equiv_{\mathcal{A}} i' \Rightarrow (q \xrightarrow{i} \Leftrightarrow q \xrightarrow{i'})$. Note that if $\mathcal{T}$ is an IOA it trivially respects $\mathcal{A}$. In these cases, we may reduce the task of the learner to learning an IA $\mathcal{H}'$ satisfying $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}' \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$. Note that $\alpha_{\mathcal{A}}(\mathcal{P})$ is a proper learning purpose for $\alpha_{\mathcal{A}}(\mathcal{T})$ since it is determinate and, by monotonicity of abstraction (Lemma 7.3.1), $\alpha_{\mathcal{A}}(\mathcal{T}) \lesssim \alpha_{\mathcal{A}}(\mathcal{P})$.

A teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ is constructed by placing a mapper component in between the teacher for $\mathcal{T}$ and the learner for $\mathcal{P}$, which translates concrete and abstract actions to each other in accordance with $\mathcal{A}$. Let $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, $\mathcal{P} = \langle I, O^\delta, P, p^0, \rightarrow_{\mathcal{P}} \rangle$, $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$, and $\mathcal{I} = \langle I, O^\delta, R, r^0, \rightarrow \rangle$. The mapper component maintains a state variable of type R, which initially is set to r^0. The behavior of the mapper component is defined as follows:

1. *Input.* If the mapper is in state r and receives an abstract input $x \in X$ from the learner, it picks a concrete input $i \in I$ such that $\Upsilon_r(i) = x$, forwards i to the teacher, and waits for a reply $\top$ or $\bot$ from the teacher. This reply is then forwarded to the learner. In case of a $\top$ reply, the mapper updates its state to the unique r' with $r \xrightarrow{i} r'$. If there is no $i \in I$ such that $\Upsilon_r(i) = x$ then the mapper returns a $\bot$ reply to the learner right away.

2. *Output.* If the mapper receives an output query Δ from the learner, it forwards Δ to the teacher. It then waits until it receives an output $o \in O^\delta$ from the teacher, and forwards $\Upsilon_r(o)$ to the learner.

3. *Reset.* If the mapper receives a **reset** from the learner, it resets its state to r^0 and forwards **reset** to the teacher.

4. *Hypothesis.* If the mapper receives a hypothesis $\mathcal{H}$ from the learner then, by Lemma 7.3.7, $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescence preserving. Since $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$, monotonicity of concretization (Lemma 7.3.4) implies $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(\mathcal{P}))$. Hence, by Lemma 7.3.6, $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$. This means that the mapper may forward $\gamma_{\mathcal{A}}(\mathcal{H})$ as a hypothesis to the teacher. If the mapper receives response **yes** from the teacher, it forwards **yes** to the learner. If the mapper receives response **no** with counterexample σo, where $\sigma = a_1 \cdots a_n$, then it constructs a run

$$(r_0, s_0) \xrightarrow{a_1} (r_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n)$$

of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ with $(r_0, s_0) = (r^0, s^0)$. It then forwards **no** to the learner, together with counterexample $z_1 \cdots z_n y$, where, for $1 \leq j \leq n$, $z_j = \Upsilon_{r_{j-1}}(a_j)$

and $y = \Upsilon_{r_n}(o)$. Finally, the mapper returns to its initial state.

The next lemma implies that, whenever the learner presents an abstract input x to the mapper, there exists a concrete input i such that $\Upsilon_r(i) = x$, and the teacher will accept input i from the mapper. So no $\perp$ replies will be sent. Moreover, whenever the teacher sends a concrete output o to the mapper, the learner will accept the corresponding abstract output $\Upsilon_r(o)$ generated by the mapper.

Lemma 7.4.1 *Let S be the maximal alternating simulation from $\mathcal{T}^\delta$ to $\mathcal{P}^\delta$. Then, for any configuration of states q, r_1 and (p, r_2) of teacher, mapper and learner, respectively, that can be reached after a finite number of steps (1)-(5) of the learning protocol, we have $(q, p) \in S$ and $(p, r_1) \sim (p, r_2)$ (here $\sim$ denotes bisimulation equivalence in $\alpha_{\mathcal{A}}(\mathcal{P})$).*

Proof By induction on the number of steps.

Initially, the teacher is in state q^0, the mapper is in state r^0, and the learner is in state (p^0, r^0). Since S relates the initial states of $\mathcal{T}^\delta$ and $\mathcal{P}^\delta$, $(q^0, p^0) \in S$. Since $\sim$ is an equivalence relation, $(p^0, r^0) \sim (p^0, r^0)$.

For the induction step, observe that after a reset or hypothesis checking step, teacher, mapper and learner all return to their initial states, which means that we reach a configuration for which, as we observed, the required properties hold. So the interesting cases are the input and output queries.

Suppose that the learner enables an abstract input $x \in X$, and takes transition $(p, r_2) \xrightarrow{x} (p', r_2')$ after presenting x to the mapper. Since $(p, r_1) \sim (p, r_2)$, there exists a transition $(p, r_1) \xrightarrow{x} (p'', r_1')$ such that $(p'', r_1') \sim (p', r_2')$. Hence, by the definition of the abstraction operator, there exists a concrete input i such that $\Upsilon_{r_1}(i) = x$, $p \xrightarrow{i} p''$ and $r_1 \xrightarrow{i} r_1'$. This means that the mapper accepts the abstract input x, forwards a corresponding concrete input, say i, to the teacher, and jumps to a new state r_1'. Since $(q, p) \in S$ and $p \xrightarrow{i} p''$, there exists a state q' such that $q \xrightarrow{i} q'$ and $(q', p'') \in S$. This means that the teacher will accept the input i from the mapper and jump to a state q'. Since $(p, r_2) \xrightarrow{x} (p', r_2')$, there exists an i' such that $\Upsilon_{r_2}(i) = x$ and $p \xrightarrow{i'} p'$. Because $\mathcal{P}$ and $\mathcal{A}$ are compatible and $i \equiv_{\mathcal{A}} i'$, $p'' \sim p'$. Hence, by Lemma 7.1.4, $(q', p') \in S$ and so the required properties hold.

Next suppose that the learner sends an output query to the mapper, which is forwarded by the mapper to the teacher. Suppose that the teacher takes transition $q \xrightarrow{o} q'$ after returning concrete output $o \in O^\delta$ to the mapper. Then the mapper jumps to the unique state r_1' with $r_1 \xrightarrow{o} r_1'$ and forwards $y = \Upsilon_{r_1}(o)$ to the learner. Since $(q, p) \in S$, there exists a state p' such that $p \xrightarrow{o} p'$ and $(q', p') \in S$. By definition of the abstraction operator, we have a transition $(p, r_1) \xrightarrow{y} (p', r_1')$. Since $(p, r_1) \sim (p, r_2)$, there exists a transition $(p, r_2) \xrightarrow{y} (p'', r_2')$ such that $(p', r_1') \sim (p'', r_2')$. This means that the learner will accept the abstract output y and jump to a state (p'', r_2'). By definition of the abstraction operator, there exists a concrete output o' such that $\Upsilon_{r_2}(o') = y$ and $p \xrightarrow{o'} p''$. Because $\mathcal{P}$ and $\mathcal{A}$ are compatible

and $o \equiv_{\mathcal{A}} o'$, $p' \sim p''$. Hence, by Lemma 7.1.4, $(q', p'') \in S$. It is straightforward to check that bisimulation is a congruence for the abstraction operator (follows also since the defining rules for $\alpha_{\mathcal{A}}$ are in the De Simone format, see [89, 40]),

that is $p' \sim p''$ implies $(p', r'_1) \sim (p'', r'_1)$. Hence, since $\sim$ is an equivalence, $(p'', r'_1) \sim (p'', r'_2)$, and so the required properties hold.

We claim that, from the perspective of a learner with learning purpose $\alpha_{\mathcal{A}}(\mathcal{P})$, a teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave exactly like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$. Since the notion of behavior has not been formalized for a teacher and a mapper, the mathematical content of this claim may not be immediately obvious. Clearly, it is routine to describe the behavior of teachers and mappers formally in some concurrency formalism, such as Milner's CCS [72] or another process algebra [19]. For instance, we may define, for each IA $\mathcal{T}$, a CCS process $\mathsf{Teacher}(\mathcal{T})$ that describes the behavior of a teacher for $\mathcal{T}$, and for each mapper $\mathcal{A}$ a CCS process $\mathsf{Mapper}(\mathcal{A})$ that models the behavior of a mapper for $\mathcal{A}$. These two CCS processes may then synchronize via actions taken from A^δ, actions Δ, δ, $\top$, $\bot$ and **reset**, and actions $\mathsf{hypothesis}(\mathcal{H})$, where $\mathcal{H}$ is an interface automaton. If we compose $\mathsf{Teacher}(\mathcal{T})$ and $\mathsf{Mapper}(\mathcal{A})$ using the CCS composition operator $|$, and apply the CCS restriction operator $\setminus$ to internalize all communications between teacher and mapper, the resulting process is observation equivalent (weakly bisimilar) to the process $\mathsf{Teacher}(\alpha_{\mathcal{A}}(\mathcal{T}))$:

$$(\mathsf{Teacher}(\mathcal{T}) \mid \mathsf{Mapper}(\mathcal{A})) \setminus L \quad \approx \quad \mathsf{Teacher}(\alpha_{\mathcal{A}}(\mathcal{T})),$$

where $L = A^\delta \cup \{\Delta, \delta, \top, \bot, \mathbf{reset}, \mathsf{hypothesis}\}$. It is in this precise, formal sense that one should read the following theorem. The reason why we do not refer to the CCS formalization in the statement and proof of this theorem is that we feel that the resulting notational overhead would obscure rather than clarify.

Theorem 7.4.2 *Let $\mathcal{T}$, $\mathcal{A}$ and $\mathcal{P}$ be as above. A teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$.*

Proof Initially, the state of the teacher for $\mathcal{T}$ is q^0 and the state of the mapper for $\mathcal{A}$ is r^0, which is consistent with the initial state (q^0, r^0) of the teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$. Suppose the current state of the teacher for $\mathcal{T}$ is q, and the current state of the mapper is r. We consider the possible interactions between the components:

- *Input.* Suppose the learner sends an abstract input $x \in X$. Using the assumption that $\mathcal{T}$ respects $\mathcal{A}$, it is easy to see that the mapper returns $\bot$ to the learner exactly if there exists no concrete input i and state q' such that $\Upsilon_r(i) = x$ and $q \xrightarrow{i} q'$. This behavior is consistent with the behavior of a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ from state (q, r),

  Now suppose that $\Upsilon_r(i) = x$, $r \xrightarrow{i} r'$, the mapper forwards i to the teacher, the teacher jumps to a state q' such that $q \xrightarrow{i} q'$, sends a reply $\top$ to the

mapper, who jumps to state r' and forwards $\top$ to the learner. This behavior is consistent with the behavior of a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ from state (q, r), which may jump to any state (q', r') such that $(q, r) \xrightarrow{x}_{\mathsf{abst}} (q', r')$.

- *Output.* Suppose the learner sends an output query Δ. The mapper will then forwards Δ to the teacher for $\mathcal{T}$. If state q is quiescent then the teacher for $\mathcal{T}$ forwards δ to the mapper, and the mapper forwards δ to the learner. This behavior is consistent with the behavior of a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ from state (q, r). If state q is not quiescent then the teacher for $\mathcal{T}$ selects a transition $q \xrightarrow{o} q'$, jumps to q' and returns o to the mapper. The mapper then forwards $\Upsilon_r(o)$ to the learner. This behavior is consistent with the behavior of a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ from state (q, r), which nondeterministically picks an output y and state (q', r) such that $(q, r) \xrightarrow{y}_{\mathsf{abst}} (q', r)$.

- *Reset.* Suppose the learner sends a **reset** command. Then the learner returns to its initial state (p^0, r^0). The mapper moves to its initial state r^0 and forwards the **reset** command to the teacher, who also returns to its initial state q^0. This behavior is consistent with the behavior of a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$ which, upon receiving a **reset**, returns to its initial state (q^0, r^0).

- *Hypothesis.* Suppose that the learner sends a hypothesis $\mathcal{H}$. The mapper will then forward $\gamma_{\mathcal{A}}(\mathcal{H})$ as a hypothesis to the teacher for $\mathcal{T}$. If the teacher for $\mathcal{T}$ answers **yes** then the mapper forwards this answer to the learner. In this case $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$ and hence, by Lemma 7.3.5, $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}$. So when the mapper forwards **yes** to the learner, this is the proper behavior for a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$.

  If the mapper receives answer **no** with a counterexample σo then, by definition of a teacher, σ is a trace of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ and o is an output enabled by $\mathcal{T}^\delta$ after σ but not by $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ after σ. So if $\sigma = a_1 \cdots a_n$, then the mapper indeed may construct a corresponding run $(r_0, s_0) \xrightarrow{a_1} (r_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_n} (r_n, s_n)$ of $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ with $(r_0, s_0) = (r^0, s^0)$. The mapper then forwards **no** to the learner, together with counterexample ρy, where $\rho = z_1 \cdots z_n$, $z_j = \Upsilon_{r_{j-1}}(a_j)$, for $1 \leq j \leq n$, and $y = \Upsilon_{r_n}(o)$. By construction, $\rho \in \mathit{Traces}(\mathcal{H}^\delta)$. Since σo is a counterexample, σo is a trace of $\mathcal{T}^\delta$. This means that we may construct a run $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n \xrightarrow{o} q_{n+1}$ of $\mathcal{T}^\delta$ with $q_0 = q^0$. Now observe that

$$(q_0, r_0) \xrightarrow{z_1} (q_1, r_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, r_n) \xrightarrow{y} (q_{n+1}, r_n)$$

  is a run of $(\alpha_{\mathcal{A}}(\mathcal{T}))^\delta$. Hence, $y \in \mathit{out}((\alpha_{\mathcal{A}}(\mathcal{T}))^\delta$ **after** $\rho)$. Since σo is a counterexample generated by the teacher for $\mathcal{T}$, o is not enabled by $(\gamma_{\mathcal{A}}(\mathcal{H}))^\delta$ after σ. In particular, state (r_n, s_n) does not enable o. This implies state s_n does not enable y. By Lemma 7.1.2, since $\mathcal{H}$ is determinate, no state of $\mathcal{H}^\delta$ reachable via trace ρ enables y. We conclude that $y \notin \mathit{out}(\mathcal{H}^\delta$ **after** $\rho)$, and so ρy is a counterexample for a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$.

Since a teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\alpha_{\mathcal{A}}(\mathcal{T})$, it follows that we have reduced the task of learning an $\mathcal{H}$ such that $\mathcal{T}$ **ioco** $\mathcal{H} \lessapprox \mathcal{P}$ to the simpler task of learning an $\mathcal{H}$ such that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H} \lessapprox \alpha_{\mathcal{A}}(\mathcal{P})$: whenever the learner receives the answer **yes** from the mapper, indicating that $\alpha_{\mathcal{A}}(\mathcal{T})$ **ioco** $\mathcal{H}$ we know, by definition of the behavior of the mapper component, that $\gamma_{\mathcal{A}}(\mathcal{H})$ is quiescent preserving and $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H})$. Moreover, by Lemmas 7.3.4 and 7.3.6, $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$.

Recall that for output-predicting abstractions, if $\mathcal{H}$ is behavior-deterministic then $\gamma_{\mathcal{A}}(\mathcal{H})$ is behavior-deterministic. This implies that, for such abstractions, provided $\mathcal{T}$ is an IOA, whenever the mapper returns an answer **yes** to the learner, $\gamma_{\mathcal{A}}(\mathcal{H})$ is in fact the unique interface automaton (up to bisimulation) that satisfies $\mathcal{T}$ **ioco** $\gamma_{\mathcal{A}}(\mathcal{H}) \lessapprox \mathcal{P}$.

7.5 Conclusion

This chapter has provided several generalizations of the framework of [1], leading to a general theory of history dependent abstractions for learning interface automata. This work establishes some very interesting links between previous work on concurrency theory, model-based testing, and automata learning.

The theory of abstractions presented in this chapter is not complete yet and deserves further study. The link between this theory and the theory of abstract interpretation [30, 31] needs to be investigated further. Also the concept of XY-simulations, which nicely generalizes three fundamental concepts from concurrency theory (bisimulations, simulations and alternating simulations), deserves further study. Finally, an obvious challenge is to generalize the theory of this chapter to SUTs that are not determinate.

Chapter 8 presents the prototype tool Tomte, which is able to automatically construct mappers for a restricted class of scalarset automata, in which one can test for equality of data parameters, but no operations on data are allowed.

Chapter 8

Counterexample-Guided Abstraction Refinement for Learning Scalarset Mealy Machines

This chapter introduces the tool Tomte, which is able to construct mappers similar to those presented in chapter 7, fully automatically for a restricted class of extended finite state machines where one can test for equality of data parameters, but no operations on data are allowed. To fulfill its task, Tomte uses counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior, the abstraction is refined automatically. This chapter is organized as follows. In section 8.1 the class of scalarset Mealy Machines is introduced and the algorithm to learn mappers for them is presented. Section 8.3 explains the abstraction learning algorithm in more detail with an example, and reports a summary of the experiments done with Tomte. Finally, the conclusions are presented in section 8.4.

8.1 The World of Tomte

The general approach for using abstraction in automata learning is phrased most naturally at a general, semantic level. And indeed, this is what was done in the previous chapter. However, if we want to devise effective algorithms and implement them, we must restrict attention to a class of automata and mappers that can be finitely represented. This section describes the class of SUTs that the Tomte tool can learn, as well as the classes of mappers and learning purposes that it uses.

8.1.1 Scalarset Mealy Machines

We assume a universe $\mathcal{V}$ of *variables*. Each variable $v \in \mathcal{V}$ has a type $\mathsf{type}(v) \subseteq \mathbb{N} \cup \{\bot\}$, where $\mathbb{N}$ is the set of natural numbers and $\bot$ denotes the undefined value. If V is a set of variables, then a *valuation* for V is a function ξ that maps each variable in V to an element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for V. We also assume a set C of *constants* which contains $\hat{\bot}$ and a function $\gamma : C \to \mathbb{N} \cup \{\bot\}$ that assigns a value to each constant. We define $\gamma(\hat{\bot}) = \bot$. If $c \in C$ is a constant then we define $\mathsf{type}(c) = \{\gamma(c)\}$. A *term* over V is either a variable or a constant, that is, an element of $C \cup V$. We write $\mathcal{T}$ for the set of all terms. If t is a term over V and ξ is a valuation for V then we write $[\![t]\!]_\xi$ for the value to which t evaluates:

$$[\![t]\!]_\xi = \begin{cases} \xi(t) & \text{if } t \in V \\ \gamma(t) & \text{if } t \in C \end{cases}$$

A *formula* or *guard* φ over V is a Boolean combination of expressions of the form $t = t'$, where t and t' are terms over V. We write $\mathcal{G}$ for the set of all formulas over $\mathcal{V}$. If ξ is a valuation for V and φ is a formula over V, then we write $\xi \models \varphi$ to denote that ξ satisfies φ. We assume a set E of *event primitives* and for each event primitive ε an arity $\mathsf{arity}(\varepsilon) \in \mathbb{N}$. An *event term* for ε is an expression $\varepsilon(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $n = \mathsf{arity}(\varepsilon)$. We write $\mathcal{ET}$ for the set of event terms.

Event signature An *event signature* Σ is a pair $\langle T_I, T_O \rangle$, where T_I and T_O are finite sets of event terms such that $T_I \cap T_O = \emptyset$ and each term in $T_I \cup T_O$ is of the form $\varepsilon(p_1, \ldots, p_n)$ with $p_1, \ldots, p_n$ pairwise different variables. We require that the event primitives as well as the variables of different event terms in $T_I \cup T_O$ are distinct. We refer to the variables occurring in an event signature as *parameters*.

Below *scalarset Mealy machines* are defined. The scalarset datatype was introduced by Ip and Dill [49] as part of their work on symmetry reduction in verification. Operations on scalarsets are restricted so that states are guaranteed to have the same future behaviors, up to permutation of the elements of the scalarsets. Using the symmetries implied by the scalarsets, a verifier can automatically generate a reduced state space. On scalarsets no operations are allowed, we only allow the use of constants in C. The only predicate symbol that may be used is equality.

Definition A *scalarset Mealy machine (SMM)* $\mathcal{M}$ is a tuple $\langle \Sigma, V, L, l_0, \Gamma \rangle$, where

- $\Sigma = \langle T_I, T_O \rangle$ is an event signature, with $\bot \notin \mathsf{type}(p)$, for each parameter p of Σ,
- $V \subseteq \mathcal{V}$ is a finite set of state variables, with $\bot \in \mathsf{type}(v)$, for each $v \in V$,
- L is a finite set of locations,

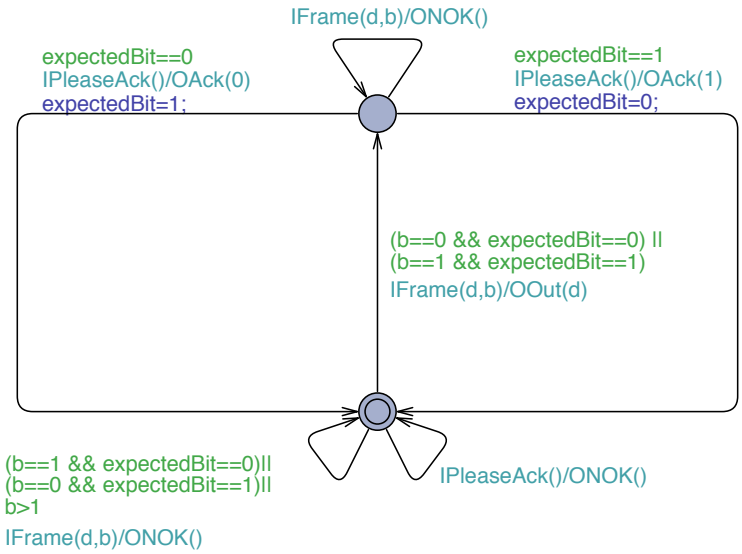

Figure 8.1: A sample SMM: alternating bit protocol receiver

- $l_0 \in L$ is the initial location,
- $\Gamma \subseteq L \times T_I \times \mathcal{G} \times (V \to \mathcal{T}) \times \mathcal{ET} \times L$ is a finite set of transitions. In a transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_l), l' \rangle \in \Gamma$, we refer to l as the *source*, g as the *guard*, ϱ as the *update* and l' as the *target*. We require that g is a formula over $V \cup \{p_1, \ldots, p_k\}$, there exists an event term $\varepsilon_O(q_1, \ldots, q_l) \in T_O$ such that, for each i, u_i is a term over V with $\mathsf{type}(u_i) \subseteq \mathsf{type}(q_i) \cup \{\bot\}$, and, for each v, $\varrho(v) \in V \cup C \cup \{p_1, \ldots, p_k\}$ and $\mathsf{type}(\varrho(v)) \subseteq \mathsf{type}(v)$.

We say that $\mathcal{M}$ is *deterministic* if, for all distinct transitions $\tau_1 = \langle l_1, e_1^I, g_1, \varrho_1, e_1^0, l_1' \rangle$ and $\tau_2 = \langle l_2, e_2^I, g_2, \varrho_2, e_2^0, l_2' \rangle$ in Γ, $l_1 = l_2$ and $e_1^I = e_2^I$ implies $g_1 \wedge g_2 \equiv \mathsf{false}$.

Example Figure 8.1 is a model of receiver running alternating bit protocol, which is a SMM.

Semantics of SMM To each SMM $\mathcal{M}$ we associate an IA $[\![\mathcal{M}]\!]$ in the obvious way. Transitions of the Mealy machine are turned into pairs of consecutive input and output transitions of the corresponding interface automaton.

The semantics of an event term $\varepsilon(p_1, \ldots, p_k)$ is the set of actions $[\![\varepsilon(p_1, \ldots, p_k)]\!] = \{\varepsilon(d_1, \cdots, d_k) \mid d_i \in \mathsf{type}(p_i), 1 \leq i \leq k\}$. The semantics of a set T of event terms is defined by pointwise extension: $[\![T]\!] = \bigcup_{e \in T} [\![e]\!]$.

Let $\mathcal{M} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a SMM with $\Sigma = \langle T_I, T_O \rangle$. The semantics of $\mathcal{M}$, denoted $[\![\mathcal{M}]\!]$, is the interface automaton $\langle I, O, Q, q^0, \to \rangle$ where

- $I = [\![T_I]\!]$ and $O = [\![T_O]\!]$,

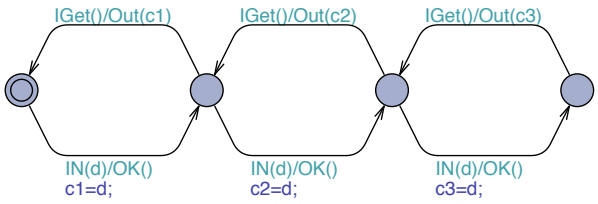

Figure 8.2: A sample SMM which is not restricted: a LIFO biffer of size 3

- $Q = Ł \times \mathsf{Val}(V) \cup Ł \times \mathsf{Val}(V) \times O$,
- $q^0 = (l_0, \xi_0)$, where $\xi_0(v) = \perp$, for $v \in V$,
- $\rightarrow \subseteq Q \times (I \cup O) \times Q$ is the smallest set that satisfies

$$\frac{\begin{array}{c}\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_\ell), l' \rangle \in \Gamma \\ \forall i \leq k, \iota(p_i) = d_i \qquad \xi \cup \iota \models g \\ \xi' = (\xi \cup \gamma \cup \iota) \circ \varrho \\ \forall i \leq \ell, [\![u_i]\!]_{\xi'} = d'_i \neq \perp\end{array}}{\begin{array}{c}(l, \xi) \xrightarrow{\varepsilon_I(d_1, \ldots, d_k)} (l', \xi', \varepsilon_O(d'_1, \ldots, d'_l)) \\ (l', \xi', \varepsilon_O(d'_1, \ldots, d'_l)) \xrightarrow{\varepsilon_O(d'_1, \ldots, d'_\ell)} (l', \xi')\end{array}}$$

Observe that if $\mathcal{M}$ is a deterministic SMM then $[\![\mathcal{M}]\!]$ is a behavior-deterministic IA. Tomte can learn the subclass of deterministic SMMs, which only record the first and the last occurrence of an input parameter:

Restricted SMMs Let $\mathcal{M}$ be a SMM. Variable v *records the last occurrence* of input parameter p of $\mathcal{M}$ if for each transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, if $p \in \{p_1, \ldots, p_k\}$ then $\varrho(v) = p$ else $\varrho(v) = v$. Moreover, v may not occur in the codomain of ϱ. Variable v *records the first occurrence* of input parameter p if for each transition $\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, e, l' \rangle \in \Gamma$, if $p \in \{p_1, \ldots, p_k\}$ and $g \Rightarrow v = \hat{\perp}$ holds then $\varrho(v) = p$ else $\varrho(v) = v$. Moreover, v may not occur in the codomain of ϱ.

We say that $\mathcal{M}$ only records the first and the last occurrence of parameters if, whenever $\varrho(v) = p$ in some transition, v either records the first or the last occurrence of p.

Example Figure 8.1, the alternating bit protocol receiver, shows a restricted SMM, because the state variables do not record any input parameter. But the SMM of figure 8.2 is not restricted, as the model records three copies of input parameter d. A restricted SMM may only record the first and the last occurrences of an input parameter, that are at most two copies.

Symmetry Reduction in Scalarset Mealy Machines

This section provides lemmas and theorems to prove the general bisimilarity of two SMM's in case of their bisimilarity when their variable domains are substituted with a large enough finite subset of natural numbers. We assume in this section that, all the variables in V and all variables of SMM's have domain $\mathbb{N} \cup \{\perp\}$.

Definition An *automorphism* is a morphism[1] from a mathematical object to itself.

Definition We call an automorphism $h : \mathbb{N} \cup \{\perp\} \to \mathbb{N} \cup \{\perp\}$ *constant respecting* if $h(\perp) = \perp$ and it maps the value of each constant in the language to itself, ie. $\forall c \in C, h(\gamma(c)) = \gamma(c)$.

Lemma 8.1.1 states that applying a constant respecting automorphism on a valuation of a set of variables preserves the logical values of the formulas under that valuation. This lemma is needed to prove that computing the value of a term under a valuation and applying a constant respecting automorphism on that value, is equivalent to computing the value of that term under the composition of the automorphism and the valuation.

Lemma 8.1.1 *If h is a constant respecting automorphism, then*

$$\xi \models \varphi \Leftrightarrow h(\xi) \models \varphi$$

where φ is a formula over V, ξ is a valuation for V, and $h(\xi)$ is the valuation for V defined by $h(\xi) = h \circ \xi$.

Proof We prove this lemma by induction on the number of operators in φ.

Basis φ has no operators, that is φ is an atomic formula. An atomic formula has four possible forms

1. $\varphi \equiv c = c'$ where $c, c' \in C$:

$$\begin{aligned} h(\xi) &\models \phi \Leftrightarrow \\ h(\xi) &\models c = c' \Leftrightarrow \\ \gamma(c) &= \gamma(c') \Leftrightarrow \\ \xi &\models c = c' \Leftrightarrow \\ \xi &\models \phi \end{aligned}$$

[1] A morphism is an abstraction derived from structure-preserving mappings between two mathematical structures. The study of morphisms and of the structures (called objects) over which they are defined, is central to category theory.

2. $\varphi \equiv v = c$ where $v \in V$ and $c \in C$:

$$\begin{aligned} h(\xi) &\models \phi \Leftrightarrow \\ h(\xi) &\models v = c \Leftrightarrow \\ h(\xi)(v) &= \gamma(c) \Leftrightarrow (h \text{ is constant respecting.}) \\ h(\xi(v)) &= h(\gamma(c)) \Leftrightarrow (h \text{ is a bijection.}) \\ \xi(v) &= \gamma(c) \Leftrightarrow \\ \xi &\models v = c \Leftrightarrow \\ \xi &\models \phi \end{aligned}$$

3. $\varphi \equiv c = v$ where $v \in V$ and $c \in C$:

$$\begin{aligned} h(\xi) &\models \phi \Leftrightarrow \\ h(\xi) &\models c = v \Leftrightarrow \\ \gamma(c) &= h(\xi)(v) \Leftrightarrow (h \text{ is constant respecting.}) \\ h(\gamma(c)) &= h(\xi(v)) \Leftrightarrow (h \text{ is a bijection.}) \\ \gamma(c) &= \xi(v) \Leftrightarrow \\ \xi &\models c = v \Leftrightarrow \\ \xi &\models \phi \end{aligned}$$

4. $\varphi \equiv v = v'$ where $v, v' \in V$:

$$\begin{aligned} h(\xi) &\models \phi \Leftrightarrow \\ h(\xi) &\models v = v' \Leftrightarrow \\ h(\xi)(v) &= h(\xi)(v') \Leftrightarrow (h \text{ is a bijection.}) \\ \xi(v) &= \xi(v') \Leftrightarrow \\ \xi &\models v = v' \Leftrightarrow \\ \xi &\models \phi \end{aligned}$$

Induction Assume that for all formulas φ with at most k operators, we have

$$\xi \models \varphi \Leftrightarrow h(\xi) \models \varphi.$$

We prove for any formula φ' with $k+1$ operators, we have

$$\xi \models \varphi' \Leftrightarrow h(\xi) \models \varphi'.$$

φ' has two possible forms:

1. $\varphi' \equiv \neg(\varphi_0)$
2. $\varphi' \equiv \varphi_1 \wedge \varphi_2$

For each possible form, we perform the induction assumption on the subformulas of φ':

1. $\varphi' \equiv \neg(\varphi_0)$

$$\begin{aligned} h(\xi) \models \varphi' &\Leftrightarrow \\ h(\xi) \models \neg(\varphi_0) &\Leftrightarrow \\ h(\xi) \not\models \varphi_0 &\Leftrightarrow (\text{ induction assumption}) \\ \xi \not\models \varphi_0 &\Leftrightarrow \\ \xi \models \neg(\varphi_0) &\Leftrightarrow \\ \xi \models \varphi' & \end{aligned}$$

2. $\varphi' \equiv \varphi_1 \wedge \varphi_2$

$$\begin{aligned} h(\xi) \models \varphi' &\Leftrightarrow \\ h(\xi) \models \varphi_1 \wedge \varphi_2 &\Leftrightarrow \\ h(\xi) \models \varphi_1 \wedge h(\xi) \models \varphi_2 &\Leftrightarrow (\text{ induction assumption}) \\ \xi \models \varphi_1 \wedge \xi \models \varphi_2 &\Leftrightarrow \\ \xi \models \varphi' & \end{aligned}$$

Lemma 8.1.2 asserts that applying a constant respecting automorphism on the value of term under a valuation, is the same as computing the value of the term under the composition of the automorphism and the valuation. This assertion is required for proving that a constant respecting automorphism preserves the structure of a SMM which is the semantics of a SMM.

Lemma 8.1.2 *If h is a constant respecting automorphism, then*

$$h([\![t]\!]_\xi) = [\![t]\!]_{h(\xi)}$$

where t is a term such that $t \in C \cup V$, and ξ is a valuation of V.

Proof There are two possible cases:

1. $t \triangleq c \in C$:

$$\begin{aligned} h([\![t]\!]_\xi) &= h([\![c]\!]_\xi) \\ &= h(\gamma(c)) \ (h \text{ is constant respecting.}) \\ &= \gamma(c) \\ &= [\![c]\!]_{h(\xi)} \\ &= [\![t]\!]_{h(\xi)} \end{aligned}$$

2. $t \triangleq v \in V$

$$\begin{aligned} h([\![t]\!]_\xi) &= h([\![v]\!]_\xi) \\ &= h(\xi(v)) \\ &= h(\xi)(v) \\ &= [\![v]\!]_{h(\xi)} \\ &= [\![t]\!]_{h(\xi)} \end{aligned}$$

Lemma 8.1.3 states that applying a constant preserving automorphism maintains the structure of a SMM which represents the semantics of a SMM. Applying a constant preserving automorphism on a SMM means applying the automorphism on the states of the SMM, that is for each state applying the automorphism on the valuation represents that state, as well as applying the automorphism on the value of the constants and terms appearing in the transitions.

Lemma 8.1.3 *Let $\mathcal{M}$ be a SMM. If h is a constant respecting automorphism, then for interface automaton $\mathcal{I} = [\![\mathcal{M}]\!]$, we have*

$$q \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)} q' \Rightarrow h(q) \xrightarrow{\varepsilon_I(h(d_1),\ldots,h(d_k))} h(q'),$$

and

$$q' \xrightarrow{\varepsilon_O(d'_1,\ldots,d'_\ell)} q'' \Rightarrow h(q') \xrightarrow{\varepsilon_O(h(d'_1),\ldots,h(d'_\ell))} h(q''),$$

where q, q' and q'' are states of $\mathcal{I}$ and have the form of (l, ξ), $(l', \xi', \varepsilon_O(d'_1, \ldots, d'_\ell))$ and (l', ξ'), respectively, and $h(q) = (l, h(\xi))$, $h(q') = (l', h(\xi'), \varepsilon_O(h(d'_1), \ldots, h(d'_\ell)))$ and $h(q'') = (l', h(\xi'))$.

Proof Suppose $q \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)} q'$, then there exists a transition

$$\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_\ell), l' \rangle$$

of $\mathcal{M}$ and a valuation ι of parameters of $\{p_1, \ldots, p_k\}$ such that if $\iota(p_i) = d_i$ for $1 \leq i \leq k$,

$$\begin{aligned} &\xi \cup \iota \models g \\ &\xi' \equiv (\xi \cup \gamma \cup \iota) \circ \varrho \end{aligned}$$

If we apply a constant respecting automorphism h, we have:

$$\iota(p_i) = d_i \Rightarrow h \circ \iota(p_i) = h(\iota)(p_i) = h(d_i) \text{ for } 1 \leq i \leq k \tag{8.1}$$

$$\begin{aligned} &\xi \cup \iota \models g \Rightarrow \text{(lemma 8.1.1)} \\ &h \circ (\xi \cup \iota) \models g \Rightarrow \\ &h(\xi) \cup h(\iota) \models g \end{aligned} \tag{8.2}$$

and

$$\begin{aligned} \xi' \equiv (\xi \cup \gamma \cup \iota) \circ \rho \Rightarrow \\ h(\xi') \equiv h \circ (\xi \cup \gamma \cup \iota) \circ \rho \Rightarrow \\ h(\xi') \equiv h(\xi \cup \gamma \cup \iota) \circ \rho \Rightarrow \\ h(\xi') \equiv (h(\xi) \cup \gamma \cup h(\iota)) \circ \rho \end{aligned} \tag{8.3}$$

For $1 \leq i \leq \ell$,

$$\begin{aligned} [\![u_i]\!]_{\xi'} = d'_i \Rightarrow \\ h([\![u_i]\!]_{\xi'}) = h(d'_i) \Rightarrow \text{(lemma 8.1.2)} \\ [\![u_i]\!]_{h(\xi')} = h(d'_i) \end{aligned} \tag{8.4}$$

From (8.1), (8.2), (8.3) and (8.4), we conclude

$$\frac{\begin{array}{c} \forall i \leq k, h(\iota)(p_i) = h(d_i) \qquad h(\xi) \cup h(\iota) \models g \\ h(\xi') = (h(\xi) \cup \gamma \cup h(\iota)) \circ \varrho \\ \forall i \leq \ell, [\![u_i]\!]_{h(\xi')} = h(d'_i) \end{array}}{\begin{array}{l} (l, h(\xi)) \xrightarrow{\varepsilon_I(h(d_1),\ldots,h(d_k))} (l', h(\xi'), \varepsilon_O(h(d'_1), \ldots, h(d'_\ell))) \\ h(q) \xrightarrow{\varepsilon_I(h(d_1),\ldots,h(d_k))} h(q') \end{array}}$$

Furthermore, there exists state $q'' = (l', \xi')$ of $\mathcal{M}$ such that

$$q' \xrightarrow{\varepsilon_O(d'_1,\ldots,d'_\ell)} q''$$

and $h(q'') = (l', h(\xi'))$. Hence,

$$h(q') \xrightarrow{\varepsilon_O(h(d_1),\ldots,h(d_\ell))} h(q'').$$

Theorem 8.1.1 is the heart of this section which proves that for bisimilarity checking of two SMM's with infinite domains, it is enough to substitute the domains by finite subsets of the domains and do the checking.

Definition For each SMM $\mathcal{S} = \langle \Sigma, V, L, l_0, \Gamma \rangle$, we define $\mathcal{S}^n$ to be the SMM obtained from $\mathcal{S}$ by replacing the types of the variables with $\{0, \ldots, n-1\} \cup \{\perp\}$. Furthermore, we define $\mathsf{Val}^n(V)$ to be the set of all valuations of V over $\{0, \ldots, n-1\} \cup \{\perp\}$.

Theorem 8.1.4 *Let $\mathcal{S}_1 = \langle \Sigma_1, V_1, L_1, l_0^1, \Gamma_1 \rangle$ and $\mathcal{S}_2 = \langle \Sigma_2, V_2, L_2, l_0^2, \Gamma_2 \rangle$ be SMM's with $\Sigma_1 = \Sigma_2$. Let n_0 be large enough (larger than the number of variables of $\mathcal{S}_1$ and $\mathcal{S}_2$ + sum of the number of parameters + number of constants) (n_0 must be larger than the value of the largest constant.) and assume $\mathcal{S}_1^{n_0} \leftrightarrows \mathcal{S}_2^{n_0}$, that is $\mathcal{S}_1^{n_0}$ and $\mathcal{S}_2^{n_0}$ are bisimilar,*

Then, $\mathcal{S}_1 \leftrightarrows \mathcal{S}_2$.

Proof Let $R \subseteq (L_1 \times \mathsf{Val}^{n_0}(V_1) \cup L_1 \times \mathsf{Val}^{n_0}(V_1) \times O) \times (L_2 \times \mathsf{Val}^{n_0}(V_2) \cup L_2 \times \mathsf{Val}^{n_0}(V_2) \times O)$ be a bisimulation from $[\![\mathcal{S}_1^{n_0}]\!]$ to $[\![\mathcal{S}_2^{n_0}]\!]$.

Define

$$R' = \{(s_1, s_2) \in (L_1 \times \mathsf{Val}(V_1) \cup L_1 \times \mathsf{Val}(V_1) \times O) \times (L_2 \times \mathsf{Val}(V_2) \cup L_2 \times \mathsf{Val}(V_2) \times O) \mid$$
$$\exists \text{ a constant respecting automorphism } h : (h(s_1), h(s_2)) \in R\}.$$

We claim that R' is a bisimulation from $[\![\mathcal{S}_1]\!]$ to $[\![\mathcal{S}_2]\!]$.

Since the initial states of $\mathcal{S}_1$ and $\mathcal{S}_2$ are also the initial states of $\mathcal{S}_1^{n_0}$ and $\mathcal{S}_2^{n_0}$, and they are related by R, they are also related by R' (take the the identity function as automorphism).

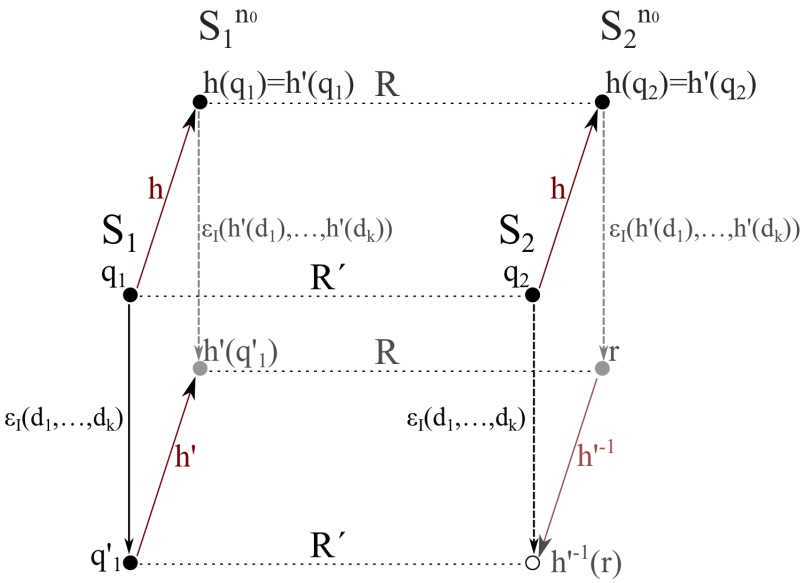

Figure 8.3: Bisimulation between finite automata versus infinite automata

Suppose q_1 and q_1' be states of $[\![\mathcal{S}_1]\!]$ such that $q_1 \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)} q_1'$, and q_2 be a state of $[\![\mathcal{S}_2]\!]$ such that $q_1 R' q_2$, Then

- There exists a constant respecting automorphism h such that

$$(h(q_1), h(q_2)) \in R.$$

- There is a transition

$$\langle l, \varepsilon_I(p_1, \ldots, p_k), g, \varrho, \varepsilon_O(u_1, \ldots, u_l), l' \rangle$$

such that $q = (l, \xi)$ and $q' = (l', \xi', \varepsilon_O(d'_1, \ldots, d'_\ell))$, where $d'_i = [\![u_i]\!]_{\xi'}$ for $1 \leq i \leq \ell$, and if ι is a valuation with $\iota(p_i) = d_i$ (for $1 \leq i \leq k$),

$$\xi \cup \iota \models g$$
$$\xi' \equiv (\xi \cup \gamma \cup \iota) \circ \varrho$$

- There exists a constant respecting automorphism h' such that

$$h'(q_1) = h(q_1)$$
$$h'(q_2) = h(q_2)$$
$$h'(d_i) \leq n_0 \text{ for } 1 \leq i \leq k$$

(Follows since n_0 is greater than $\mid V_1 \mid + \mid V_2 \mid + \mid C \mid + \mid P \mid$)

- By lemma 8.1.3,

$$h'(q_1) \xrightarrow{\varepsilon_I(h'(d_1),\ldots,h'(d_k))} h'(q'_1).$$

Since $(h'(q_1), h'(q_2)) \in R$, and R is a bismulation, there exists a state r such that

$$h'(q_2) \xrightarrow{\varepsilon_I(h'(d_1),\ldots,h'(d_k))} r$$

and

$$(h(q'_1), r) \in R.$$

By the lemma 8.1.3, since h'^{-1} is a constant respecting automorphism such that,

$$q_2 \xrightarrow{\varepsilon_I(d_1,\ldots,d_k)} h'^{-1}(r).$$

Observe

$$(q'_1, h'^{-1}(r)) \in R'.$$

This construction is depicted in figure 8.3.

Using lemma 8.1.3, case of ε_O is similar.

8.1.2 Abstraction Table

For each event signature, we define a family of symbolic abstractions, parametrized by what is called an *abstraction table*. For each parameter p, an abstraction table contains a list of variables and constants. If v occurs in the list for p then, intuitively, this means that for the future behavior of the SUT it may be relevant whether p equals v or not.

Abstraction table Let $\Sigma = \langle T_I, T_O \rangle$ be an event signature, let P be the set of parameters that occur in T_I, and let U be the set of parameters that occur in T_O. Let v_p^f and v_p^l be fresh variables with $\mathsf{type}(v_p^f) = \mathsf{type}(v_p^l) = \mathsf{type}(p) \cup \{\bot\}$, and let $V^f = \{v_p^f \mid p \in P\}$ and $V^l = \{v_p^l \mid p \in P\}$. An *abstraction table* for Σ is a

function $F : P \cup U \to (V^f \cup V^l \cup C)^*$, such that, for each $p \in P \cup U$, all elements of sequence $F(p)$ are distinct, and, for each $p \in U$, $F(p)$ lists all the elements of $V^f \cup V^l \cup C$.

Mapper induced by abstraction table Let $\Sigma = \langle T_I, T_O \rangle$ be a signature and let F be an abstraction table for Σ. Let P be the set of parameters in T_I and let U be the set of parameters in T_O. Then the mapper $\mathcal{A}_\Sigma^F = \langle \mathcal{I}, X, Y, \Upsilon \rangle$ with $\mathcal{I} = \langle I, O^\delta, R, r^0, \to \rangle$ is defined as follows:

- Let, for $p \in P \cup U$, p' be a fresh variable with $\mathsf{type}(p') = \{0, \ldots, |F(p)|-1\} \cup \{\bot\}$. Let $T_X = \{\varepsilon(p'_1, \ldots, p'_k) \mid \varepsilon(p_1, \ldots, p_k) \in T_I\}$ and $T_Y = \{\varepsilon(p'_1, \ldots, p'_l) \mid \varepsilon(p_1, \ldots, p_l) \in T_O\}$. Then $I = [\![T_I]\!]$, $O = [\![T_O]\!]$, $X = [\![T_X]\!]$, and $Y = [\![T_Y]\!]$.
- $R = \mathsf{Val}(V^f \cup V^l)$ and $r^0(v_p^f) = r^0(v_p^l) = \bot$, for all $p \in P$.
- $\to$ and Υ are defined as follows, for all $r \in R$,
  1. $r \xrightarrow{\delta} r$ and $\Upsilon_r(\delta) = \delta$.
  2. Let $o = \varepsilon_O(d_1, \ldots, d_k)$ and let $\varepsilon_O(q_1, \ldots, q_k) \in T_O$. Then $r \xrightarrow{o} r$ and $\Upsilon_r(o) = \varepsilon_O(d'_1, \ldots, d'_k)$, where, for $1 \le j \le k$, d'_j is the smallest index m such that $[\![F(q_j)_m]\!]_r = d_j$, or $d'_j = \bot$ if there is no such index.
  3. Let $i = \varepsilon_I(d_1, \ldots, d_k)$ and let $\varepsilon_I(p_1, \ldots, p_k) \in T_I$. Let $r_0 = r$ and, for $1 \le j \le k$,

$$r_j \quad = \quad \begin{cases} r_{j-1}[d_j/v_{p_j}^f][d_j/v_{p_j}^l] & \text{if } r_{j-1}(v_{p_j}^f) = \bot \\ r_{j-1}[d_j/v_{p_j}^l] & \text{otherwise} \end{cases} \tag{8.5}$$

     Then $r \xrightarrow{i} r_k$ and $\Upsilon_r(i) = \varepsilon_I(d'_1, \ldots, d'_k)$, where, for $1 \le j \le k$, d'_j is the smallest index m such that $[\![F(p_j)_m]\!]_{r_{j-1}} = d_j$, or $d'_j = \bot$ if there is no such index.

Strictly speaking, the mappers $\mathcal{A}_\Sigma^F$ introduced above are not output-predicting. In fact, in each state r of the mapper there are infinitely many concrete outputs that are mapped to the abstract output $\bot$. However, SUTs whose behavior can be described by scalarset Mealy machines have a remarkable property: the only possible values for output parameters are constants and values of previously received inputs. As a result, the mapper will never send an abstract output with a parameter $\bot$ to the learner. This in turn implies that in the behavior-deterministic hypothesis $\mathcal{H}$ generated by the learner $\bot$ will not occur as an output parameter. Since $\mathcal{A}_\Sigma^F$ is output-predicting for all the other outputs, it follows that the concretization $\gamma_{\mathcal{A}_\Sigma^F}(\mathcal{H})$ is behavior-deterministic.

Theorem 8.1.5 *Let $\mathcal{M} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a SMM that only records the first and last occurrence of parameters. Let F be an abstraction table for Σ. Then $\alpha_{\mathcal{A}_\Sigma^F}([\![\mathcal{M}]\!])$ is finitary.*

Proof (sketch) Let S be the relation that deemes two states s_1, s_2 of $\alpha_{\mathcal{A}_\Sigma^F)}([\![\mathcal{M}]\!])$ equivalent iff there exists a constant respecting automorphism h with $h(s_1) = s_2$. Then, clearly, S is an equivalence relation. We claim that S is a bisimulation on $\alpha_{\mathcal{A}_\Sigma^F}([\![\mathcal{M}]\!])$. Suppose that $(s_1, s_2) \in S$ with $s_1 = (q_1, r_1)$, $s_2 = (q_2, r_2)$, and h a constant respecting automorphism that maps s_1 to s_2. Suppose further that $(q_1, r_1) \xrightarrow{z} (q_1', r_1')$. Then, by definition of the abstraction operator, there exists an a such that $q_1 \xrightarrow{a} q_1'$, $r_1 \xrightarrow{a} r_1'$, and $\Upsilon_r(a) = z$. By Lemma 8.1.3, $h(q_1) \xrightarrow{h(a)} h(q_1')$. Moreover, by the definition of mapper $\mathcal{A}_\Sigma^F$, $h(r_1) \xrightarrow{h(a)} h(r_1')$. Also by definition of mapper $\mathcal{A}_\Sigma^F$, $\Upsilon_r(a) = \Upsilon_{h(r)}(h(a))$. We conclude $(h(q_1), h(r_1)) \xrightarrow{z} (h(q_1'), h(r_1'))$, that is, $h(s_1) \xrightarrow{z} h(s_2)$. Hence, S is a bisimulation, as claimed.

To each state $s = ((l, \xi), r)$ or $s = ((l, \xi, o), r)$ of $\alpha_{\mathcal{A}_\Sigma^F}([\![\mathcal{M}]\!])$ we associate a partial equivalence relation $\mathsf{PER}(s)$ on $V \cup V^f \cup V^l \cup C$ which puts variables or constants in the same equivalence class whenever they evaluate to the same non-$\perp$ value:

$$\mathsf{PER}(s) \quad = \quad \{\{t' \in V \cup V^f \cup V^l \cup C \mid [\![t']\!]_{\xi \cup r} = [\![t]\!]_{\xi \cup r} \neq \perp\} \mid t \in V \cup V^f \cup V^l \cup C\}.$$

The reader may check that $s' = h(s)$ implies that $\mathsf{PER}(s) = \mathsf{PER}(s')$. Since in all (reachable) states of the form $(l, \xi, \epsilon_O(d_1', \ldots, d_l'))$ the values of the output parameters are determined by the event term $\epsilon_O(u_1, \ldots, u_l)$ and valuation ξ, it follows that bisimulation S has finitely many equivalence classes. Hence the induced quotient structure, which is behaviorally equivalent to $\alpha_{\mathcal{A}_\Sigma^F)}([\![\mathcal{M}]\!])$, is a finite interface automaton and. $\alpha_{\mathcal{A}_\Sigma^F)}([\![\mathcal{M}]\!])$ is finitary.

Theorem 8.1.6 *Let $\mathcal{M} = \langle \Sigma, V, L, l_0, \Gamma \rangle$ be a deterministic SMM that only records the first and last occurrence of parameters. Let $\mathsf{Full}(\Sigma)$ be the abstraction table F for Σ in which, for each p, $F(p)$ has maximal length. Then $\alpha_{\mathcal{A}_\Sigma^{\mathsf{Full}(\Sigma)}}([\![\mathcal{M}]\!])$ is behavior deterministic.*

Proof (sketch) Suppose a state s of $\alpha_{\mathcal{A}_\Sigma^{\mathsf{Full}(\Sigma)}}([\![\mathcal{M}]\!])$ has two distinct outgoing transitions. Then s must be of the form $s = ((l, \xi), r)$. Suppose s has outgoing transitions $s \xrightarrow{x} s'$ and $s \xrightarrow{x} s''$. Then s' and s'' can only be different because in x some abstract parameter has value $\perp$, leading to different concrete values in s' and s''. But since these concrete values are fresh, there exists a constant preserving automorphism h such that $h(s') = s''$. Hence, by the proof of the previous theorem, s' and s'' are bisimilar, as required.

Definition Let I and O be disjoint sets of input and output actions. Then $\mathsf{Mealy}(I, O)$ is the IA $\langle I, O, \{m_0, m_1\}, m_0, \{(m_0, i, m_1) \mid i \in I\} \cup \{(m_1, o, m_0) \mid o \in O\} \rangle$.

Lemma 8.1.7 *Let $\Sigma = \langle T_I, T_O \rangle$ be an event signature, let $I = [\![T_I]\!]$ and $O = [\![T_O]\!]$, let $\mathcal{M}$ be a SMM over Σ, let F be an abstraction table for Σ, and let $\mathcal{P} = \mathsf{Mealy}(I, O)$. Then $\mathcal{A}_\Sigma^F$ and $\mathcal{P}$ are compatible, $[\![\mathcal{M}]\!]$ **ioco** $\mathcal{P}$ and $\gamma_{\mathcal{A}_\Sigma^F}(\alpha_{\mathcal{A}_\Sigma^F}(\mathcal{P})) \lessapprox \mathcal{P}$.*

We have now solved the problem of learning deterministic symbolic Mealy machines that only record the first and last occurrence of parameters, at least in theory! By Theorem 8.1.6 and Lemma 8.1.7, we can apply the approach described in Section 7.3 if we use mapper $\mathcal{A}_\Sigma^{\mathsf{Full}(\Sigma)}$ and learning purpose $\mathsf{Mealy}(I, O)$. By Theorem 8.1.5, we know that $\mathcal{I} = \alpha_{\mathcal{A}_\Sigma^{\mathsf{Full}(\Sigma)}}(\llbracket \mathcal{M} \rrbracket)$ is finite state and so if we use our mapper component in combination with any tool that is able to learn finite interface automata, the learning procedure will always terminate. The only problem is that in practice $\mathcal{I}$ will be much too large. For instance, if we have an event signature with just 10 parameters including an event type with 4 parameters, then the number of actions of $\mathcal{I}$ will be at least $21^4 \approx 2.10^5$, which is far beyond what state-of-the-art learning tools can handle.

8.2 Counterexample-Guided Abstraction Refinement

In order to avoid the practical problems that arise with the abstraction table $\mathsf{Full}(\Sigma)$, we take an approach based on counterexample-guided abstraction. We start with the simplest mapper, which is induced by the abstraction table F with $F(p) = \epsilon$, for all $p \in P$, and only refine the abstraction (i.e., add an element to the table) when we have to. Our CEGAR procedure starts with the simplest of these abstractions (essentially the empty table). If using this abstraction we find a correct hypothesis we are done. Otherwise, we refine the abstraction by adding an entry to our table. Since there are only finitely many possible abstractions and we know that the abstraction that corresponds to the full table is sound, our CEGAR approach will always terminate (at least in theory).

The reason why refinement steps may be necessary is that $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket M \rrbracket)$ may exhibit nondeterministic behavior. During the construction of a hypothesis we will not observe nondeterministic behavior, even when table F is not full: due to our choice of the concretization function υ, which always chooses fresh values, the mapper induced by F will behave exactly as the mapper induced by $\mathsf{Full}(\Sigma)$, except that the set of abstract actions is smaller. Only if the learner has formulated a hypothesis $\mathcal{H}$, the mapper has forwarded this hypothesis to the teacher, and the teacher responds with **no** with a counterexample $\sigma 0$ we may face a problem: the counterexample may be due to the fact that $\mathcal{H}$ is incorrect, but it may also be due to the fact that $\alpha_{\mathcal{A}_\Sigma^F}(\llbracket M \rrbracket)$ is not behavior-deterministic. In order to figure out the nature of the counterexample, we first construct the unique execution of $\mathcal{A}_\Sigma^F$ with trace σo. Then we assign a color to each occurrence of a parameter value in this execution:

Definition Let $r \xrightarrow{i} r'$ be a transition of $\mathcal{A}_\Sigma^F$ with $i = \varepsilon_I(d_1, \ldots, d_k)$ and let $\varepsilon_I(p_1, \ldots, p_k) \in T_I$. Let $\Upsilon_r(i) = \varepsilon_I(d'_1, \ldots, d'_k)$. Then we say that a value d_j is *green* if $d'_j \neq \perp$. Value d_j is *black* if $d'_j = \perp$ and d_j equals the value of some constant

or occurs in the codomain of state r_{j-1} (where r_{j-1} is defined as in equation (8.5) above). Value d_j is *red* if it is neither green nor black.

Intuitively, a value of an input parameter p is green if it equals a value of a previous parameter or constant that is listed in the abstraction table, a value is black if it equals a previous value that is not listed in the abstraction table, and a value is red if it is fresh. The mapper now does a new experiment on the SUT in which all the black values of input parameters in the trace are converted into fresh "red" values. If, after abstraction, the trace of the original counterexample and the outcome of the new experiment are the same, then hypothesis $\mathcal{H}$ is incorrect and we forward the abstract counterexample to the learner. But if they are different then we may conclude that $\alpha_{\mathcal{A}^F_\Sigma}(\mathcal{T})$ is not behavior-deterministic. In this case, the run for the original counterexample contains at least one black value, which determines a new entry that we can add to the abstraction table.

8.2.1 Implementation details

In Tomte, LearnLib is employed to do the automata inference. The Tomte tool, which implements the mapper component, sits in between LearnLib and the SUT.

Rather than using a separate model-based testing tool to test the correctness of hypotheses, we used the ability of LearnLib to generate test sequences. Once a hypothesis has been constructed, LearnLib can generate long test sequences to check if the hypothesis is correct, using a library of well-known test generation algorithms. Depending on whether LearnLib is constructing a hypothesis or is testing one, Tomte adjusts its behavior. During the learning phase, Tomte selects fresh concrete values whenever it receives an abstract action with parameter value $\perp$. During the testing phase, instead of selecting fresh concrete values for an abstract parameter value $\perp$, random values are selected. In this way, we ensure that the full concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is explored. By tuning the probability distribution used by Tomte for selecting random values, we obtained an efficient and reliable way to test the correctness of $\gamma_{\mathcal{A}}(\mathcal{H})$: in none of our experiments we suffered from false positives.

Once Tomte has discovered that the current abstraction is too coarse, it must select a black valued parameter and "make it green" by adding it as a new entry to the abstraction table. This is done via a series of experiments in which black values are converted one by one into fresh values, until a change in observable output is detected.

The algorithm for finding this new abstraction is outlined in Algorithm 1. Here, for an occurrence b, $\mathsf{param}(b)$ gives the corresponding formal parameter, $\mathsf{source}(b)$ gives the previous occurrence b' which, according to the execution of $\mathcal{A}^F_\Sigma$, is the source of the value of b, and $\mathsf{variable}(b)$ gives the variable in which the value of b is stored in the execution of $\mathcal{A}^F_\Sigma$. To keep the presentation simple, the set of constants here is assumed to be empty. A series of experiments in which black occurrences and their sources are converted one by one into fresh values (lines 4

and 5) is run on the SUT (lines 6 and 7), until a change in observable output is detected (lines 8 and 9). When the new abstraction entry has been added to the abstraction table, the learner is restarted with the new abstract alphabet.

Algorithm 1 Abstraction refinement

Input: Counterexample $c = i_1 \cdots i_n$
Output: Pair (p, v) with v new entry for $F(p)$ in abstraction table
1: Add black occurrences of values in c to queue Q
2: **while** abstraction not found **do**
3: b:=dequeued black value occurrence from Q
4: c′:=c, where b is set to a fresh value
5: c″:=c, where $\mathsf{source}(b)$ is set to a fresh value
6: o′:=output from running c′ on SUT
7: o″:=output from running c″ on SUT
8: **if** o′ and o″ are different from output of c **then**
9: **return** $(\mathsf{param}(b), \mathsf{variable}(\mathsf{source}(b)))$
10: **end if**
11: **end while**

8.3 Experiments

In this section the operation of Tomte is illustrated by means of the Session Initiation Protocol (SIP) [85]. SIP is an application layer protocol for controlling multimedia communication sessions, such as voice and video calls over Internet Protocol (IP). The protocol can be used for creating, modifying and terminating two-party (unicast) or multiparty (multicast) sessions. Sessions may consist of one or several media streams.

In this section, SIP protocol is referred as presented in [1], and Tomte is used to construct the abstraction for infering the behavior of the SIP Server entity when setting up connections with a SIP Client. The input messages from the SIP Client to the SIP Server are represented as *Method(From, To, Contact, CallId, CSeq, Via)*, where

- *Method* defines the type of request, either INVITE, PRACK, or ACK,
- *From* and *To* are addresses of the originator and receiver of the request,
- *CallId* is a unique session identifier,
- *CSeq* is a sequence number that orders transactions in a session,
- *Contact* is the address where the Client wants to receive input messages, and

- *Via* indicates the transport path that is used for the transaction.

The output messages from the SIP Server to the SIP Client are represented as *StatusCode(From,To,CallId,CSeq,Contact,Via)*, where *StatusCode* is a three digit status code that indicates the outcome of a previous request from the Client, and the other parameters are as for a input message.

Tomte expects the input messages to begin with an "*I*", and the output messages to begin with an "*O*". Initially, no abstraction for the input is defined in the learner, which means all parameter values are $\perp$. As a result every parameter in every input action is treated in the same way and the mapper selects a fresh concrete value, e.g. the abstract input trace *IINVITE*($\perp$, $\perp$, $\perp$), *IACK*($\perp$, $\perp$, $\perp$), *IPRACK*($\perp$, $\perp$, $\perp$), *IPRACK*($\perp$, $\perp$, $\perp$) is translated to the concrete trace *IINVITE(1, 2, 3), IACK(4, 5, 6), IPRACK(7, 8, 9), IPRACK(10, 11, 12)*. In the learning phase queries with distinct parameter values are sent to the SUT, so that the learner constructs the abstract Mealy machine shown in Figure 8.4. In the testing phase parameter values may be duplicated, which may lead to non-deterministic behavior. The test trace *IINVITE, IACK, IPRACK, IPRACK* in Figure 8.5 leads to an *0200* output that is not foreseen by the hypothesis, which produces an *O481*.

Rerunning the trace with distinct values as before leads to an *O481* output. Thus, to resolve this problem, the input abstraction must be refined. Therefore, we identify the green and black values in the trace and try to remove black values. The algorithm first successfully removes black value No. 1 by replacing the nine in the *IPRACK* input with a fresh value and observing the same output as before. However, removing black edge No. 2 changes the final outcome of the trace to an *O481* output. As a result, we need to refine the input abstraction by adding an equality check between the first parameter of the last *IINVITE* message and the first parameter of an *IPRACK* message to every *IPRACK* input. Apart from refining the input alphabet, every concrete output parameter value is abstracted to either a constant or a previous occurrence of a parameter. The abstract value is the index of the corresponding entry in the abstraction table. After every input abstraction refinement, the learning process needs to be restarted. We proceed until the learner finishes the inference process without getting interrupted by a non-deterministic output.

Besides SIP protocol, Tomte was successfully used to learn the following models:

- Alternating bit protocol (ABP) [13], see figure 8.1.
- Biometric Passport [2]
- A simple Login System
- Farmer-Wolf-Goat-Cabbage Puzzle
- Palindrome/Repeated Digit Checker

All models are available at `http://www.italia.cs.ru.nl/tools/`.

Table 8.1 gives an overview of the systems learned, with the number of input refinement steps, total learning and testing queries, number of states of the learned abstract model, and time needed for learning and testing (in seconds). All models inferred have been checked to be bisimilar to their SUT. For this purpose the learned model is combined with the abstraction and the CADP tool set, `http://www.inrialpes.fr/vasy/cadp/`, is used for equivalence checking.

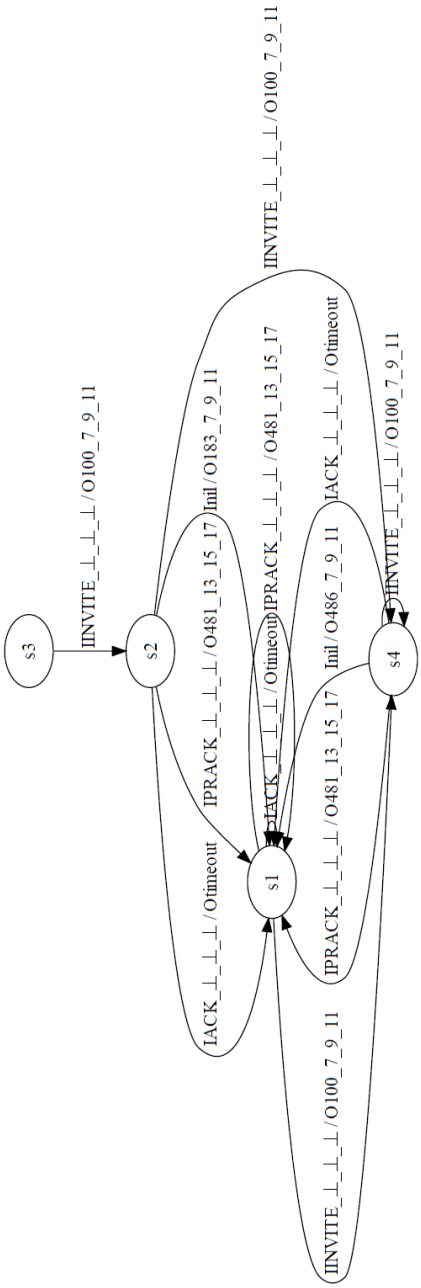

Figure 8.4: Hypothesis of SIP protocol

IINVITE				O100			IACK			timeout	IPRACK			O481			IPRACK			O200		
p1	p2	p3	o1	o2	o3	q1	q2	q3		r1	r2	r3	t1	t2	t3	r1	r2	r3	u1	u2	u3
⊥	⊥	⊥	7	9	11	⊥	⊥	⊥		⊥	⊥	⊥	13	15	17	⊥	⊥	⊥	13	15	17
16	17	9	16	17	9	4	10	25		9	3	22	9	3	22	16	15	21	16	15	21

Figure 8.5: Non-determinism in SIP protocol

System under test	Input refinements	Learning/Testing queries	States	Learning/Testing time
Alternating Bit Protocol - Sender	1	193/3001	7	1.3s/104.9s
Alternating Bit Protocol - Receiver	2	145/3002	4	0.9s/134.5s
Alternating Bit Protocol - Channel	0	31/3000	2	0.3s/107.5s
Biometric Passport	3	2199/3582	5	7.7s/94.5s
Session Initiation Protocol	3	1755/3402	13	8.3s/35.9s
Login	3	639/3063	5	2.0s/56.8s
Farmer-Wolf-Goat-Cabbage Puzzle	4	699/3467	10	4.4s/121.8s
Palindrome/Repdigit Checker	11	3461/3293	1	10.3s/256.4s

Table 8.1: Learning statistics

8.4 Conclusion

Tomte implements an algorithm which uses the technique of counterexample-guided abstraction refinement: initially, the algorithms starts with a very course abstraction $\mathcal{A}$, which is subsequently refined if it turns out that $\alpha_{\mathcal{A}}(\mathcal{T})$ is not behavior-deterministic. The idea to use CEGAR for learning state machines has been explored recently by Howar at al [47] who developed and implemented a CEGAR procedure for the special case in which the abstraction is static and does not depend on the history.

Tomte is able to construct mappers for a restricted class of extended transition systems, called scalarset automata. In scalarset automata, one can test for equality of data parameters, but no operations on data are allowed. Scalarsets also motivated the recent work of [23], which establishes a canonical form for a variation of scalarset automata.

Even though the class of systems to which our approach currently applies is limited, the fact that we are able to learn models of systems with data fully automatically is a major step towards a practically useful technology for automatic learning of models of software components.

Currently, Tomte can learn SUTs that may only remember the last and first occurrence of a parameter. Apparently, it is easy to dispose this restriction. Furthermore, the CEGAR based approach of this chapter can be further extended to systems that may apply simple or known operations on data, using technology for automatic detection of likely invariants, such as Daikon [34].

Chapter 9

Conclusion of Part Two

During the last two decades, important developments have taken place in the area of automata learning, see e.g. [6, 84, 83, 45, 18, 64, 46, 71]. History dependent abstraction operators are the key for scaling existing methods for active learning of automata to realistic applications. A major challenge is the development of algorithms for the automatic construction of abstraction mappers: the availability of such algorithms will boost the applicability of automata learning technology. In paper [47], a method is presented that is able to automatically construct certain state-free mappers. Aarts, Jonsson & Uijen [1] have proposed a framework for history dependent abstraction operators. Using this framework they succeeded to automatically infer models of several realistic software components with large state spaces, including fragments of the TCP and SIP protocols. Despite this success, the approach of Aarts et al suffers from limitations that hinder its applicability in practice.

Chapter 7 of this dissertation provided several generalizations of the framework of [1], leading to a general theory of history dependent abstractions for learning interface automata. This theory offers four important improvement to the theory of history dependent abstraction operators:

From Mealy machines to interface automata The approach of [1] is based on Mealy machines, in which each input induces exactly one output. In practice, however, inputs and outputs often do not alternate: a single input may sometimes be followed by a series of outputs, sometimes by no output at all, etc. For this reason, the approach of chapter 7 is based on interface automata [32], which have separate input and output transitions, rather than the more restricted Mealy machines.

Learning purposes In practice, it is often neither feasible nor necessary to learn a model for the complete behavior of the SUT. Typically, it is better to concentrate the learning efforts on certain parts of the state space. This is achieved using the

concept of a *learning purpose* [3] (known as *test purpose* within model-based testing theory [87, 51, 99]), which allows one to restrict the learning process to relevant interaction patterns only. In the theory of chapter 7, the concept of a mapper component of [1] is integrated with the concept of a learning purpose of [3]. This integration is nontrivial and constitutes one of the main technical contributions of this thesis.

Forgetful abstractions The main result of [1] only applies to abstractions that are output predicting. This means that no information gets lost and we infer a model that is behaviorally equivalent to the model of the teacher: $\mathcal{M} \approx \gamma_{\mathcal{A}}(\mathcal{H})$. In order to deal with the complexity of real systems, we need to support also forgetful abstractions that over-approximate the behavior of the teacher. For this reason, in this thesis, the notion of equivalence $\approx$ is replaced by the **ioco** relation, which is one of the main notions of conformance in model-based black-box testing [92, 93] and closely related to the alternating simulations of [5].

Handling equivalence queries Active learning algorithms in the style of Angluin [6] alternate two phases. In the first phase a hypothesis is constructed and in the second phase (called an *equivalence query* by Angluin [6]) the correctness of this hypothesis is checked. In general, no guarantees can be given that the answer to an equivalence query is correct. Tools such as LearnLib, "approximate" equivalence queries via long test sequences, which are computed using some established algorithms for model-based testing of Mealy machines. In the approach of [1], one needs to answer equivalence queries of the form $\alpha_{\mathcal{A}}(\mathcal{M}) \approx \mathcal{H}$. In order to do this, a long test sequence for $\mathcal{H}$ that is computed by the learner is concretized by the mapper. The resulting output of the SUT is abstracted again by the mapper and sent back to the learner. Only if the resulting output agrees with the output of $\mathcal{H}$ the hypothesis is accepted. This means that the outcome of an equivalence query depends on the choices of the mapper. If, for instance, the mapper always picks the same concrete action for a given abstract action and a given history, then it may occur that the test sequence does not reveal any problem, even though $\alpha_{\mathcal{A}}(\mathcal{M}) \not\approx \mathcal{H}$. Hence the task of generating a good test sequence is divided between the learner and the mapper, with an unclear division of responsibilities. This makes it extremely difficult to establish good coverage measures for equivalence queries. A more sensible approach, which is elaborated in this thesis, is to test whether the concretization $\gamma_{\mathcal{A}}(\mathcal{H})$ is equivalent to $\mathcal{M}$, using state-of-the-art model based testing algorithms for systems with data, and to translate the outcomes of that experiment back to the abstract setting.

The theoretical advances that are described in chapter 7 of this thesis are important to bring automata learning tools and techniques to a level where they can be used routinely in industrial practice.

Chapter 8 of this thesis presented a prototype tool Tomte, which automatically constructs mappers for a restricted class of extended transition systems, using a

counterexample-guided abstraction refinement approach.

The CEGAR technique is used for learning state machines in a recent research of Howar at al [47], who developed and implemented a CEGAR procedure for the special case in which the abstraction is static and does not depend on the execution history. The approach of this thesis, however, is applicable to a much richer class of systems, which for instance includes the SIP protocol and the various components of the Alternating Bit Protocol.

Chapter 10

Epilogue

This thesis was organized in two parts following two different research areas, specifically, verification of wireless sensor networks and automata learning.

Part one covered my research in 2008 and 2009, when I was working in collaboration with the European project Quasimodo to devise a CEGAR-based method for verification of an arbitrary size wireless sensor network provided by Chess. This research led to considerable results, namely establishing the necessary and sufficient conditions, in the form of constraints on the parameters of a fully-connected WSN in order to guarantee its being synchronized, and discovering a flaw in Chess implementation. Despite the original plan, we did not succeed to use CEGAR approach for conquering the state space explosion problem when verifying the Chess synchronization protocol for a general network of arbitrary size.

Part two covered my research in 2010 and 2011 when I collaborated in design and implementation of a CEGAR-based algorithm for automatically learning a limited class of parametric systems, called scalarset symbolic Mealy machines. Furthermore, we provided a solid theoretical foundation for learning interface automata using a large class of abstractions.

In sum, CEGAR-based parametric model checking of WSN synchronization protocols is difficult, if possible at all. Nevertheless, this challenge is worth more research. CEGAR seems to be a powerful method in extending the available approaches to automata learning.

Bibliography

[1] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[2] F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.

[3] F. Aarts and F.W. Vaandrager. Learning I/O automata. In P. Gastin and F. Laroussinie, editors, *21st International Conference on Concurrency Theory (CONCUR), Paris, France, August 31st - September 3rd, 2010, Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[4] R. Alur and D.L. Dill. The theory of timed automata. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer-Verlag, 1992.

[5] R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[7] F.A. Assegei. Decentralized frame synchronization of a TDMA-based wireless sensor network. Master's thesis, Eindhoven University of Technology, Department of Electrical Engineering, 2008.

[8] B. Ayari, N. Ben Hamida, and B. Kaminska. Automatic test vector generation for mixed-signal circuits. In *Proceedings of the 1995 European conference on Design and Test*, page 458. IEEE Computer Society, 1995.

[9] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[10] C. Baier and J.P. Katoen. *Principles of model checking*. MIT Press, 2008.

[11] R. Bakhshi, F. Bonnet, W. Fokkink, and B. Haverkort. Formal analysis techniques for gossiping protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):28–36, 2007.

[12] R. Bakhshi, L. Cloth, W. Fokkink, and B.R. Haverkort. Mean-field analysis for the evaluation of gossip protocols. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 247–256. IEEE Computer Society, 2009.

[13] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[14] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA, pages 125–126. IEEE Computer Society, 2006.

[15] G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[16] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer-Verlag, July/August 1996.

[17] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaala tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243, 1996.

[18] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

[19] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.

[20] G. M. Brown and L. Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2006.

[21] N.G. de Bruijn. A survey of the project Automath. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*. Academic Press, 1980.

[22] A. Carioni, S. Ghilardi, and S. Ranise. MCMT in the land of parameterized timed automata. In J. Giesl and R. Hähnle, editors, *6th International Verification Workshop (VERIFY 2010), associated with IJCAR, Edinburgh, UK, July 20-21, 2010. Proceedings*, 2010. Available at `http://homes.dsi.unimi.it/~ghilardi/allegati/verify.pdf`.

[23] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *ATVA*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[24] D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of MANET simulators. In *Proceedings of the 2002 Workshop on Principles of Mobile Computing, POMC 2002, October 30-31, 2002, Toulouse, France*, pages 38–43. ACM, 2002.

[25] SJ Chandra and JH Patel. A hierarchical approach test vector generation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 495–501. ACM, 1987.

[26] W.T. Cheng and T.J. Chakraborty. Gentest: an automatic test-generation system for sequential circuits. *Computer*, 22(4):43–49, 1989.

[27] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In E. Al-Shaer, A.D. Keromytis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 426–439. ACM, 2010.

[28] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[29] P.M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125. IEEE Computer Society, 2009.

[30] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symposium on Principles of programming Languages*, pages 238–252, 1977.

[31] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[32] L. de Alfaro and T.A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26 of *Software Engineering Notes*, pages 109–120, New York, September 2001. ACM Press.

[33] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[34] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[35] R. Fan and N.A. Lynch. Gradient clock synchronization. *Distributed Computing*, 18(4):255–266, 2006.

[36] A. Fehnker, M. Fruth, and A. McIver. Graphical modelling for simulation and formal analysis of wireless network protocols. In M. Butler, C.B. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2009.

[37] A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the LMAC protocol for wireless sensor networks. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*, pages 253–272. Springer, 2007.

[38] G.L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, 2003.

[39] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In J. Giesl and R. Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 22–29. Springer, 2010.

[40] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[41] K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using Uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, pages 2–13. IEEE Computer Society, 1997.

[42] F. Heidarian. A comment on Assegei's use of Kalman filters for clock synchronization, October 2010. ICIS, Radboud University Nijmegen. Available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/HSV09/`.

[43] F. Heidarian, J. Schmaltz, and F.W. Vaandrager. Analysis of a clock synchronization protocol for wireless sensor networks. In A. Cavalcanti and D. Dams, editors, *Proceedings 16th International Symposium of Formal Methods (FM2009), Eindhoven, the Netherlands, November 2-6, 2009*, volume 5850 of *Lecture Notes in Computer Science*, pages 516–531. Springer, 2009.

[44] T.A. Henzinger and J. Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32–40, 2007.

[45] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, April 2010.

[46] F. Howar, B. Steffen, and M. Merten. From ZULU to RERS. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010.

[47] F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2011.

[48] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W.A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[49] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[50] A. Jantsch. *Modeling embedded systems and SoCs: concurrency and time in models of computation*. Morgan Kaufmann Pub, 2004.

[51] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.

[52] B. Jonsson. Modular verification of asynchronous networks. In PODC'87 [78], pages 152–166.

[53] A.-M. Kermarrec and M. van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, 2007.

[54] M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*, pages 322–323. IEEE Computer Society, 2004.

[55] M. Lajolo, M. Rebaudengo, MS Reorda, M. Violante, and L. Lavagno. Behavioral-level test vector generation for system-on-chip designs. In *High-Level Design Validation and Test Workshop, 2000. Proceedings. IEEE International*, pages 21–26. IEEE, 2000.

[56] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[57] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[58] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[59] E.A. Lee. Absolutely positively on time: what would it take?[embedded computing systems]. *Computer*, 38(7):85–87, 2005.

[60] E.A. Lee. The future of embedded software. In *ARTEMIS Conference, Graz, http://ptolemy.eecs.berkeley.edu/presentations/06/FutureOfEmbeddedSoftware Lee Graz.ppt*, 2006.

[61] E.A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72, May*, 2007.

[62] E.A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[63] C. Lenzen, T. Locher, and R. Wattenhofer. Clock synchronization with bounded global and local skew. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 509–518. IEEE Computer Society, 2008.

[64] M. Leucker. Learning meets verification. In F.S. de Boer, M. M. Bonsangue, S. Graf, and W.P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2006.

[65] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[66] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[67] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In PODC'87 [78], pages 137–151. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[68] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[69] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[70] L. Meier and L. Thiele. Gradient clock synchronization in sensor networks. Technical Report 219, Computer Engineering and Networks Laboratory, ETH Zurich, 2005.

[71] M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In P.A. Abdulla and K.R.M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.

[72] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[73] U. Montanari and M. Pistore. Checking Bisimilarity for Finitary pi-Calculus. In I. Lee and S.A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 1995.

[74] E.F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 129–153. Princeton University Press, 1956.

[75] M.E.J. Newman. Detecting community structure in networks. *The European Physical Journal B*, 38:321–330, 2004.

[76] O. Niese. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, 2003.

[77] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[78] *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, August 1987.

[79] R.M. Pussente and V.C. Barbosa. An algorithm for clock synchronization with the gradient property in sensor networks. *Journal of Parallel and Distributed Computing*, 69(3):261 – 265, 2009.

[80] QUASIMODO. Preliminary description of case studies, January 2009. Deliverable 5.2 from the FP7 ICT STREP project 214755 (QUASIMODO).

[81] QUASIMODO. Dissemination and exploitation, February 2010. Deliverable 5.6 from the FP7 ICT STREP project 214755 (QUASIMODO).

[82] QUASIMODO. Final report: case studies and tool integration, April 2011. Deliverable 5.10 from the FP7 ICT STREP project 214755 (QUASIMODO).

[83] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

[84] R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences (extended abstract). In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, 15-17 May 1989, Seattle, Washington, USA*, pages 411–420. ACM, 1989.

[85] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol. *Internet Engineering Task Force (IETF): RFC 3261*, 2002.

[86] J. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 304–313, New York, NY, USA, 1994. ACM.

[87] V. Rusu, K. du Bousquet, and T. Jéron. An approach to symbolic test generation. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 1-3, 2000, Proceedings*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2000.

[88] J. Schmaltz. A formal model of clock domain crossing and automated verification of time-triggered hardware. In J. Baumgartner and M. Sheeran, editors, *Formal methods in computer aided design*, pages 223–230. IEEE Computer Society, 2007.

[89] R. de Simone. Higher-level synchronising devices in MEIJE–SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[90] B. Sundararaman, U. Buy, and A. D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281 – 323, 2005.

[91] R. Tjoa, K.L. Chee, P.K. Sivaprasad, S.V. Rao, and J.G Lim. Clock drift reduction for relative time slot TDMA-based sensor networks. In *Proceedings of the* 15^{th} *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC2004)*, pages 1042–1047, September 2004.

[92] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17:103–120, 1996.

[93] J. Tretmans. Model-based testing with labelled transition systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[94] C.H. Tsai, F.D. Guo, J.H. Hong, and C.W. Wu. IEEE Std. 1149.1 boundary scan circuit capable of built-in self-testing, October 29 1996. US Patent 5,570,375.

[95] S. Tschirner, L. Xuedong, and W. Yi. Model-based validation of qos properties of biomedical sensor networks. In L. de Alfaro and J. Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 69–78. ACM, 2008.

[96] S. Umeno. Event order abstraction for parametric real-time system verification. In L. de Alfaro and J. Palsberg, editors, *EMSOFT*, pages 1–10. ACM, 2008.

[97] F.W. Vaandrager and A.L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing Journal*, 18(4):433–458, December 2006.

[98] M. Veanes and N. Bjørner. Input-output model programs. In M. Leucker and C. Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 2009.

[99] R.G. de Vries and J. Tretmans. Towards Formal Test Purposes. In E. Brinksma and J. Tretmans, editors, *Formal Approaches to Testing of Software – FATES'01*, number NS-01-4 in BRICS Notes Series, pages 61–76, University of Aarhus, Denmark, 2001. BRICS.

[100] F. van der Wateren. Personal communication, April 2009.

[101] F. Wiedijk. The "de Bruijn factor", 2010. Webpage accessed October 2010, `http://www.cs.ru.nl/~freek/factor/`.

[102] T.A.C. Willemse. Heuristics for ioco-based test-based modelling. In L. Brim, B.R. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers*, volume 4346 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2007.

[103] M. Woehrle, K. Lampka, and L. Thiele. Exploiting timed automata for conformance testing of power measurements. In J. Ouaknine and F. W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, volume 5813 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2009.

Samenvatting

Studies op het Gebied van Verificatie van Draadloze Sensornetwerken en het Construeren van Abstracties voor het Leren van Automaten

Ingebouwde systemen zijn computersystemen die zijn ingebouwd in apparaten en de functionaliteit van deze apparaten voor een belangrijk deel bepalen. Voorbeelden zijn te vinden in smartphones, moderne autos, MRI-scanners en in de zelfscanners in supermarkten. Ingebouwde systemen zijn veelal onzichtbaar, maar hebben desondanks een enorme invloed op de manier waarop de moderne mens de fysieke wereld om zich heen waarneemt en er mee interacteert. Voor ingebouwde systemen waarbij de veiligheid van mensen in het geding is, zoals autos en medische apparatuur, is de betrouwbaarheid van evident belang. Maar ook in niet-kritieke toepassingen, zoals spelcomputers en dvd-spelers, kunnen onverwachte fouten desastreus zijn voor de reputatie van een fabrikant en de verkoop van nieuwe apparaten drastisch verminderen.

Bij het ontwerpen van een betrouwbaar ingebouwde systeem moeten we typisch aantonen dat dit systeem nooit in bepaalde gevaarlijke toestanden kan komen. Verificatie en validatie (V&V) zijn hierbij cruciaal. Deze dissertatie benadert V&V vanuit twee gezichtspunten. In het eerste deel beschrijven we de modellering en verificatie van een realistische casus op het gebied van draadloze sensornetwerken. In het tweede deel onderzoeken we een geheel nieuwe techniek waarbij abstractieverfijning gebruikt wordt voor het automatisch leren van modellen van ingebouwde systemen. Deze modellen kunnen dan vervolgens ingezet worden voor V&V.

In het eerste deel van dit proefschrift onderzoeken we de toepasbaarheid van bestaande verificatiemethoden aan de hand van een industrile casus op het gebied van draadloze sensornetwerken die is aangedragen door het Nederlandse bedrijf Chess eT International BV. Een draadloos sensornetwerk (DSN) bestaat uit een verzameling kleine apparaatjes, knopen genaamd, die verbonden zijn via een netwerk en gezamenlijk een taak uitvoeren. Iedere knoop is in staat berekeningen uit te voeren (gebruikmakend van een of meer micro-controllers, processoren of DSP

chips), beschikt over wat geheugen (programma-, data- en/of flashgeheugen), over een RF transceiver (normaliter met een bi-directionele antenne), een stroombron (bijvoorbeeld batterijen of zonnecellen), en verscheidene sensoren en actuatoren. De Chess casus heeft betrekking op de regels (het protocol) dat knopen gebruiken om met elkaar te communiceren. Een effectief protocol voor draadloze sensornetwerken moet weinig stroom verbruiken, moet voorkomen dat knopen tegelijk berichten versturen (door elkaar heen praten), moet gemplementeerd kunnen worden met zo min mogelijk code en geheugen, moet voldoende bandbreedte leveren voor een gegeven applicatie, en moet om kunnen gaan met wisselende radiofrequenties en netwerkomstandigheden. Chess heeft een DSN platform ontwikkeld op basis van een epidemisch communicatiemodel. Om aan de strikte vereisten ten aanzien van energieverbruik te kunnen voldoen maakt Chess gebruik van een zogenaamd Time Division Multiple Access (TDMA) protocol, waarbij de knopen slechts een fractie van de tijd actief zijn en zich gedurende de rest van de tijd in een slaaptoestand bevinden waarin ze vrijwel geen energie gebruiken. Voor de goede werking van dit protocol is het cruciaal dat alle knopen gelijk actief zijn: het heeft geen zin wanneer een knoop berichten verstuurt wanneer al zijn buren slapen. Dit vereist dat de klokken van aangrenzende knopen (vrijwel) gelijk lopen. In dit proefschrift wordt de model checker Uppaal gebruikt om twee kloksynchronisatiealgoritmen voor het draadloze sensornetwerk van Chess te modelleren en te verifiren. Met behulp van Uppaal is het ons gelukt om foutscenarios te vinden die tot toestanden leiden waarin het netwerk niet meer gesynchroniseerd is. Middels experimenten met een draadloos sensornetwerk hebben medewerkers van Chess aangetoond dat deze scenarios zich ook daadwerkelijk kunnen voordoen. Op basis van de door ons gevonden foutscenarios introduceren we drie condities waaraan de protocolparameters moeten voldoen om fouten uit te sluiten. Met behulp van bewijstechnieken die gebruik maken van invarianten bewijzen we dat deze condities noodzakelijk en afdoende zijn voor correctheid in het speciale geval van netwerken waarin alle knopen direct met elkaar verbonden zijn. Deze bewijzen zijn gecheckt met behulp van de bewijsassistent Isabelle/HOL.

Het tweede deel van dit proefschrift beschrijft hoe tegenvoorbeeld-gedreven abstractieverfijning gebruikt kan worden om geheel automatisch modellen (toestandsdiagrammen) te leren van computersystemen puur op basis van testen en observaties van het extern waarneembare gedrag. Bestaande programmas voor het actief leren van toestandsautomaten zijn in staat om modellen te leren met maximaal circa 10.000 toestanden. Dit is ontoereikend voor het leren van modellen van realistische softwarecomponenten die, door het gebruik van programmavariabelen en dataparameters in berichten, dikwijls een veel groter aantal toestanden hebben. Abstractie blijkt cruciaal voor het leren van modellen van dergelijke systemen. In praktische toepassingen waarbij leertechnologie gebruikt word om modellen te construeren van softwarecomponenten, definiren gebruikers dikwijls handmatig abstracties waarbij een groot aantal concrete berichten worden afgebeeld op een beperkt aantal abstracte berichten. In deze dissertatie wordt een complete theorie

van abstracties voor het leren van toestanddiagrammen gepresenteerd. Er wordt aangetoond dat zulke abstracties volledig automatisch geconstrueerd kunnen worden voor een bepaalde klasse van toestanddiagrammen waarin getest kan worden op gelijkheid van data parameters, maar geen bewerkingen op data toegestaan zijn. Bij de constructie wordt gebruik gemaakt van tegenvoorbeeld-gedreven abstractieverfijning: indien een abstractie te grof is en non-deterministisch gedrag veroorzaakt in het geleerde model, dan wordt deze abstractie automatisch verfijnd. Met behulp van een prototype implementatie van ons algoritme zijn wij er in geslaagd modellen van verschillende realistische software componenten, zoals het biometrische paspoort en het SIP protocol, volledig automatisch te leren.

Summary

Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference

Embedded systems are redefining how we perceive and interact with the physical world. While mission-critical embedded applications raise obvious reliability concerns, unexpected or premature failures in even noncritical applications such as game boxes and portable video players can erode a manufacturer's reputation and greatly diminish acceptance of new devices. The design of reliable systems requires assuring that the system never moves through a dangerous state, and *verification* and *validation* (V & V) is the key. This dissertation approaches V & V of embedded systems from two different perspectives: in the first part, modeling and verification of a real-world case-study provided by the Chess eT International B.V. is described, whereas the second part investigates automata learning (automatic modeling of systems) using abstraction refinement.

In part one, the industrial case-study of Chess on wireless sensor networks is investigated. A *wireless sensor network* (WSN) is a collection of nodes organized into a cooperative network. Each node has a processing capability (one or more micro-controllers, CPUs or DSP chips), may contain multiple types of memory (program, data and flash memories), has a RF transceiver (usually with a single omnidirectional antenna), has a power source (e.g., batteries and solar cells), and accommodate various sensors and actuators. An effective protocol for wireless sensor networks must consume little power, avoid collisions, be implemented with a small code size and memory requirements, be efficient for a single application, and be tolerant to changing radio frequency and networking conditions. Chess eT International B.V. has developed a WSN platform using a gossip (epidemic) communication model. In order to meet strict energy constraints, Chess used a Time Division Multiple Access (TDMA) protocol in which the nodes are active only in a limited period and for the remainder of the time, nodes switch to an energy saving mode. In the first part of this thesis, the model checker Uppaal is used for modeling and verification of two synchronization algorithms for wireless sensor networks. Indeed, Uppaal is used for extracting the error scenarios representing the situations where the network goes out of synch. The error scenarios are repro-

ducible in reality. Based on such error scenarios, three conditions are introduced for a fully connected network to work correctly. The conditions are proved to be necessary and sufficient using invariant proof techniques. Isabelle/HOL supports the proofs.

Part two describes how counterexample-guided abstraction refinement(CEGAR) can be employed to infer models automatically through observations and test, that is, through black-box reverse engineering. State-of-the-art tools for active learning of state machines are able to learn state machines with at most in the order of 10.000 states. This is not enough for learning models of realistic software components which, due to the presence of program variables and data parameters in events, typically have much larger state spaces. Abstraction is the key when learning behavioral models of realistic systems. Hence, in most practical applications where automata learning is used to construct models of software components, researchers manually define abstractions which, depending on the history, map a large set of concrete events to a small set of abstract events that can be handled by automata learning tools. In the second part of this thesis, a full theory of abstraction for learning interface automata is presented. Moreover, it is shown how such abstractions can be constructed fully automatically for a class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. This aim is reached through counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior in the learned model, the abstraction is refined automatically. Using a prototype implementation of the algorithm, models of several realistic software components, including the biometric passport and the SIP protocol were learned fully automatically.

Curriculum Vitae

Faranak Heidarian was born in Shahrekord, Iran, on August 24, 1981. In 1999, she graduated from Farzangan Education Center in Shahrekord, and went to Sharif University of Technology in Tehran, Iran. She received her bachelor degree in Software Engineering in 2004 and her Masters degree in Computer Science in 2007. In January 2008 she went to the Netherlands to continue her education. She became a PhD student in Radboud University Nijmegen, under supervision of Frits Vaandrager.

Titles in the IPA Dissertation Series since 2006

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of

Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of

Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof as-*

sistant for Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Fac-

ulty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model coevolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06